UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

UMA LINGUAGEM CENTRADA EM *FRAMES* PARA DESENVOLVIMENTO DE SISTEMAS BASEADOS NO CONHECIMENTO

Dissertação submetida à Universidade Federal de Santa Catarina como requisito parcial à obtenção do grau de Mestre em Engenharia Elétrica

LEANDRO JOSÉ KOMOSINSKI

Florianópolis, março de 1990.

UMA LINGUAGEM CENTRADA EM FRAMES PARA DESENVOLVIMENTO DE SISTEMAS BASEADOS NO CONHECIMENTO

LEANDRO JOSÉ KOMOSINSKI

ESTA DISSERTAÇÃO FOI JULGADA ADEQUADA PARA OBTENÇÃO DO TÍTULO DE

MESTRE EM ENGENHARIA ELÉTRICA

ESPECIALIDADE CONTROLE E AUTOMAÇÃO INDUSTRIAL E APROVADA EM SUA FORMA FINAL PELO PROGRAMA DE PÓS-GRADUAÇÃO

Prof. Hervé Eric Garnousset, Dr.
ORIENTADOR

Prof. José Carlos M. Bermudez, Ph.D.
COORDENADOR DO CURSO

BANCA EXAMINADORA

Prof. Hervé Eric Garnousset, Dr.
PRESIDENTE

Prof. Pedro Manoel da Silveira, Ph.D.

Prof. Jean-Marie Farines, Dr.Ing.

Eng. Paulo Neves de Lacerda, M.Sc.

À João e Lionira, meus pais.

AGRADECIMENTOS

À Hervé Garnousset pela amizade e habilidade na orientação deste trabalho.

Ao pessoal do LCMI pelo ótimo ambiente de trabalho.

À Paulo Lacerda pelas suas importantes opiniões.

Aos colegas Maria Marta Leite, Carmen Lacerda e José Eduardo de Lucca pelo apoio e, principalmente, pela grande amizade estabelecida.

À Veroni Valenti pelo carinho e força insubstituíveis que sempre inspiraram-me.

Sumário

RESUMO

Nesta dissertação é apresentada LOF (Lisp Orientado a *Frames*), uma extensão da linguagem Le_Lisp, orientada à geração de sistemas baseados em conhecimento.

Os objetos básicos manipulados pela linguagem são os *frames* e as instâncias. Um *frame* é um meta-objeto que descreve a estrutura e o comportamento de suas instâncias. Os *frames* representam classes de objetos enquanto que suas instâncias representam indivíduos pertencentes às classes em questão.

A declaração de um *frame* é feita através da definição da estrutura (facetas) e do comportamento (*demons*) dos *slots* (atributos) que o compõe.

O principal processo de inferência em LOF é a instanciação. Este processo consiste em realizar um casamento (*matching*) entre o modelo representado por um *frame* e uma situação observada. A instância gerada neste processo é consistente com as especificações definidas no *frame*.

A linguagem LOF proporciona ao programador, além do paradigma de programação orientado a *frames*, os paradigmas funcional (através da linguagem Le_Lisp) e orientado a objetos (através da "degeneração" da estrutura do *frame*).

ABSTRACT

This dissertation presents LOF, a frame-based extention of the Le_Lisp language aimed at designing knowledge-based systems.

The fundamental objects manipulated by the LOF language are frames and instances. A frame is a meta-object that determinates the structure and the behavior of a set of individual objects which are called its instances.

Defining a frame means specifying the structure (facets) and behavior (demons) of the declarative set of its slots.

Instantiation is the principal inference mecanism in LOF. It consists in matching a frame-model against an observed situation. The instance generated in this process is consistent with frame's specifications.

LOF programmers dispose of three programming paradigms: frame-oriented, functional (based on LISP) and object-oriented (based on degenerated frame structure).

Capítulo 1

INTRODUÇÃO

Com o desenvolvimento cada vez maior da informática (tanto a nível de *hardware* quanto de *software*) uma grande variedade de problemas passaram a ser tratados por computadores. Os recentes avanços em computação sugerem que o processamento simbólico (ao invés do numérico) será a base para a próxima geração de computadores [WAH 89]. O projeto japonês para construção de computadores de quinta geração, que está fortemente baseado em conceitos de Inteligência Artificial (IA), exemplifica esta tendência. Verifica-se também o crescimento do número de sistemas que utilizam técnicas de IA, iniciado na década de 80.

Nos sistemas de IA orientados à resolução de problemas é dada muita ênfase ao papel do **conhecimento** e, por esta razão, são chamados de Sistemas Baseados em Conhecimento (SBC) (*Knowledge-Based Systems*). Os SBC são caracterizados por manterem uma **base de conhecimentos explícita** (e em geral **declarativa**) onde são armazenadas as informações a respeito de um domínio específico [BRACHMAN 85b]. A ênfase sobre o conhecimento nos SBC fez com que a Representação do Conhecimento (RC) assumisse um papel fundamental para o desenvolvimento de sistemas [BARR 81], [ALTY 84], [FIKES 85], [WOODS 86], [WAH 89].

As pesquisas em RC levaram ao desenvolvimento de vários Formalismos de Representação do Conhecimento (FRC). Estes formalismos são formados por estruturas de dados (usadas para armazenar informações) e por procedimentos (usados para inferir novas informações a partir daquelas). Eles ganharam popularidade principalmente com o surgimento, a partir da metade da década de 70, dos Sistemas Especialistas (SE).

Os SE são sistemas que armazenam e utilizam o conhecimento de um especialista humano em um domínio específico. Dentre os sistemas mais populares pode-se listar: PROSPECTOR, um sistema especialista em prospecção mineral, MYCIN, um sistema

especialista no diagnóstico e tratamento de infecção bacteriana e DENDRAL, um sistema especialista na determinação de estruturas topológicas de compostos orgânicos por análise espectográfica.

Os primeiros SE desenvolvidos foram construídos visando tratar de um único domínio. Com o tempo esta limitação foi superada e passou-se a desenvolver também ambientes de desenvolviemnto de sistemas baseados no conhecimento (conhecidos como *shells* ou geradores de SE). Estes sistemas têm a característica de dispor de um ou mais formalismos para a representação do conhecimento bem como mecanismos de inferência associados.

Devido a natureza essencialmente simbólica do processamento existente nos SE, a linguagem LISP é, juntamente com PROLOG, bastante utilizada para construí-los [ALTY 84], [WATERMAN 86], [WINSTON 89]. Em princípio, as estruturas de listas e de átomos são flexíveis para representar simbolicamente o conhecimento. As maiores vantagens atribuídas a LISP são relacionadas à facilidade de programação proporcionada por um ambiente interativo (quando interpretado), a fácil prototipagem, as estruturas poderosas de controle e a existência de boas ferramentas para depuração de programas [WINSTON 89], [RICH 83]. Embora questões relativas ao controle de SE possam ser tratadas adequadamente em LISP, esta linguagem apresenta dificuldades para a representação de conhecimentos [WATERMAN 86].

Na década de 80 a Programação Orientada a Objetos (POO) apareceu como um importante paradigma de programação. Este paradigma está baseado na construção de sistemas compostos por objetos que se comunicam através da troca de mensagens e são organizados em hierarquias. Estas características permitem a modularidade e facilitam a reutilização do *software*. Várias linguagens (como Flavors, Loops, ObjectLisp, CLOS, etc.) foram criadas de modo a incorporar as vantagens do estilo de progamação orientado a objeto ao estilo funcional de LISP produzindo, assim, uma linguagem híbrida multi-paradigma [BOBROW 86].

Dentre os vários formalismos de representação do conhecimento existentes, o modelo baseado em *frames* [MINSKY 75] é o que mais assemelha-se, tanto estruturalmente quanto por sua metodologia de representação de informações, ao sparadigma de programação orientada a objetos. Isto sugere os benefícios obtidos pela combinação deste formalismo com a linguagem LISP. O resultado é uma linguagem híbrida que combina os paradigmas funcional, orientado a *frames* e orientado a objetos.

Nesta dissertação é apresentada a linguagem LOF (Lisp Orientado a *Frames*). LOF é uma extensão da linguagem LISP orientada a resolução de problemas. O formalismo de representação do conhecimento é centrado em *frames*. Por outro lado, a semelhança entre a programação orientada a *frames* e a objeto levou a introduzir este paradigma de programação em LOF com um custo limitado.

No capítulo 2 são introduzidos alguns conceitos sobre Representação do Conhecimento (RC). Estes conceitos fundamentam vários dos formalismos de RC utilizados nos Sistemas Especialistas. Em particular, é mostrado o formalismo baseado em *frames* e como ele pode ser utilizado.

No capítulo 3 é apresentada a linguagem LOF, enfatizando-se as suas características estruturais de representação e os mecanismos de inferência que proporciona.

No capítulo 4 é feita uma comparação entre os formalismos de representação centrado em *frames* e a programação orientada a objetos. É mostrado também como LOF pode ser utilizada como linguagem de programação orientada a objetos.

No capítulo 5 é apresentado um exemplo de utilização de LOF na geração de um SE para diagnóstico de defeitos em equipamentos de Subestações de Alta Tensão.

No capítulo 6 são apresentadas algumas conclusões sobre o trabalho desenvolvido e perspectivas para sua continuação.

Capítulo 2

REPRESENTAÇÃO DO CONHECIMENTO

2.1 Introdução

Nos Sistemas Especialistas (SE) o conhecimento tem papel fundamental tanto na qualidade da solução retornada como na complexidade computacional para sua obtenção. A Representação do Conhecimento (RC) é uma área de pesquisa em Inteligência Artificial (IA) onde estuda-se o tipo de conhecimento manipulado nestes sistemas e as representações adequadas para sua utilização na resolução de tais problemas. O objetivo neste capítulo é apresentar as idéias a respeito desta área e a influência resultante no desenvolvimento de Sistemas Especialistas.

Neste capítulo são apresentados conceitos sobre SE e RC, enfatizando-se a estreita ligação existente entre eles. A seguir é introduzido o formalismo de representação do conhecimento centrado em *frames*, juntamente com uma descrição da sua utilização em alguns sistemas existentes.

2.2 Sistemas Especialistas e Representação do Conhecimento

A evolução dos sistemas de resolução de problemas deu-se em três etapas distintas [WATERMAN 86]. Na primeira, ocorrida na década de 60, as pesquisas em IA eram levadas com o propósito de construir-se sistemas que implementassem métodos genéricos de resolução de problemas. Acreditava-se, que a partir de métodos simples e de computadores poderosos (com altas capacidade de processamento e de armazenamento) pudessem ser desenvolvidos sistemas com desempenho comparável ao dos seres humanos. Porém, os resultados obtidos foram muito limitados [RICH 83]. Constatou-se então a importância do conhecimento no processo de resolução dos problemas. A segunda etapa

surge, já nos anos 70, com a perspectiva de construir-se estruturas de representação do conhecimento que possam abrigar um conhecimento suficientemente geral para permitir resolver uma grande parte de problemas reais (o sistema KRL [BOBROW 77b] é um exemplo desta família de sistemas). No entanto, a complexidade e diversidade do conhecimento envolvido na resolução de problemas reais mostrou as limitações desta abordagem [HAYES-ROTH 83]. A terceira etapa, iniciada na metade da década de 70, surge com o enfoque voltado para sistemas limitados à áreas bem restritas de conhecimento onde os formalismos de representação tornam-se apenas um mecanismo auxiliar, mas não sem importância, para a resolução de problemas. É neste contexto que surgem os SE.

Os SE tornaram-se então Sistemas Baseados em Conhecimento (SBC) (*Knowledge-Based Systems*). Sua principal característica é a separação do conhecimento sobre o domínio tratado do resto do sistema (informações de controle sobre a utilização daquele conhecimento). Além disso, tal conhecimento é armazenado explicitamente. Daqui por diante, os termos SE e SBC serão usados indistintamente.

A arquitetura básica de um SBC é composta por uma Base de Conhecimento (BC), onde são armazenadas informações acerca do domínio de atuação do sistema, e por uma Máquina de Inferência (MI) onde são definidos os mecanismos de manipulação das informações presentes na BC de modo a inferir novas informações. O desenvolvimento de um SBC consiste em extrair o conhecimento de um especialista e codificá-lo na BC. Assim, "a acumulação e codificação do conhecimento é um dos aspectos mais importantes do desenvolvimento de um SE" [WATERMAN 86].

Pelo exposto acima, o poder de resolução de um SE está diretamente ligado ao nível do conhecimento que contém. É preciso então usar formalismos capazes de representar todas as informações existentes no domínio tratado pelo SE de forma explícita e, de preferência, declarativa. Além disso, o formalismo deve facilitar os processos de inferência existentes.

As pesquisas em Representação do Conhecimento visam especificar formalismos capazes de representar informações e desenvolver procedimentos para utilizá-los. O seu desenvolvimento deve levar em conta o **poder de expressão, a legibilidade** e a **modularidade** do formalismo em questão aliados a **eficiência computacional** dos mecanismos de inferência associados. O poder de expressão diz respeito à capacidade do formalismo poder representar os conhecimentos existentes na formulação do problema. A legibilidade está ligada à naturalidade com que informações são representadas e a facilidade da sua compreensão. A modularidade está associada à capacidade de incluir, alterar e remover informações de forma simples e sem comprometer a consistência das informações resultantes. A eficiência computacional diz respeito à necessidade do sistema gerar inferências em tempos aceitáveis, isto é, cuja ordem de magnitude seja compatível com a dinâmica do meio físico onde deve atuar o sistema.

Os mecanismos de inferência estão diretamente ligados ao formalismo escolhido para representar o conhecimento. No entanto o comportamento inteligente de um formalismo está ligado à sua capacidade de explorar eficientemente as informações representadas e não somente a sua eficiência inferencial.

Não existe uma teoria sobre a Representação do Conhecimento em virtude da complexidade inerente a este assunto que envolve questões computacionais, psicológicas e filosóficas [BARR 86]. No entanto, vários formalismos foram desenvolvidos e usados em Sistemas Especialistas produzindo um resultado satisfatório. Dentre os formalismos mais conhecidos estão a **lógica**, as **regras de produção**, as **redes semânticas** e os **frames**. Uma descrição detalhada sobre estes formalismos pode ser encontrada em [BARR 86].

A representação através da lógica está fundamentada numa teoria consistente. A resolução de problemas é encarada como uma prova de teorema, feita a partir de um conjunto de axiomas que representam as informações conhecidas. A ineficiência deste formalismo está ligada a total independência do processo de dedução em relação ao conhecimento sendo utilizado (tornando a dedução "insensível" a informações de controle

particulares ao problema tratado) e na sua incapacidade em representar naturalmente estruturas complexas de conhecimento como, por exemplo, informações *default*.

As regras de produção são o formalismo mais utilizado hoje em dia nos SE. A sua principal característica é a simplicidade da estrutura de representação, feita através de regras da forma "SE condições são satisfeitas ENTÃO ações devem ser executadas". As regras são um meio usado naturalmente pelos especialistas para expressar o seu conhecimento [WINSTON 84]. Informações heurísticas são facilmente representadas através de regras. Um SE é composto por um conjunto de regras independentes cuja execução é controlada por uma máquina de inferência responsável pelo disparo em cadeia destas regras. O formalismo apresenta no entanto deficiências devido ao baixo desempenho da máquina de inferência, no decréscimo da legibilidade da BC quando o número de regras do sistema aumenta e na falta de estruturação do conhecimento devido a uniformidade sintática das regras.

Nas redes semânticas o conhecimento é representado através de um grafo, onde os nós descrevem conceitos sobre objetos, eventos ou atividades e as ligações etiquetadas definem relações semânticas entre eles. As inferências estão baseadas em algoritmos de procura em grafos que são auxiliados pela semântica associadas às ligações. A deficiência deste formalismo está associada à falta de estruturação do grafo devido a inexistência de um consenso sobre a semântica dos nós, além das dificuldades computacionais inerentes aos processos de procura em grafos.

Os *frames* estão baseados no mesmo princípio das redes semânticas, isto é, uma representação que consiste de nós e ligações. Porém, os nós são fortemente estruturados e a semântica das ligações é limitada às relações de pertinencia e de inclusão. Todas as informações a respeito de um objeto estão agrupadas em uma única estrutura de dados. Este modelo foi definido por Minsky [MINSKY 75] para representar situações estereotipadas. Associado aos *frames* estão vários tipos de informação como a estrutura e o comportamento de suas instâncias. A principal característica dos *frames* é, portanto, a capacidade de representar conceitos complexos. O processo de inferência fundamental

neste formalismo é o casamento entre o modelo representado pelo *frame* e uma situação observada. Este formalismo é descrito em detalhes na seção seguinte.

2.3 O Formalismo de Frames

Os *frames* são uma modelagem da teoria psicológica que afirma que muitas das atividades cognitivas humanas estão baseadas no processo de reconhecimento de situações [BARR 86]. Apesar de serem relativamente vagos os conceitos definidos por Minsky, os *frames* foram usados como fundamento para muitos sistemas como, por exemplo KL-ONE [BRACHMAN 85], KEE [INTELLICORP 86, 87], PIP [JOHNSON 86], KRL [BOBROW 77], CENTAUR [AIKINS 81].

A falta de precisão no artigo seminal de Minsky [MINSKY 75] proporcionou uma grande variedade de interpretações sobre a estrutura e o papel dos *frames* dentro de um sistema de resolução de problemas. Consequentemente, são poucos os conceitos comuns existentes nos sistemas desenvolvidos e a terminologia varia de um sistema para outro.

Os trabalhos relativos aos *frames* dividem-se em duas linhas: uma, de cunho epistemológico, investiga a construção de estruturas capazes de representar as informações complexas existentes em linguagem natural. O sistema KL-ONE é um representante típico desta linha. A outra linha de pesquisa, mais pragmática, investiga a construção de estruturas adequadas à resolução de problemas particulares. Os sistemas PIP [JOHNSON 86] e CENTAUR [AIKINS 83] são exemplos desta linha de trabalho. Atualmente, os *frames* começam a despertar maior interesse como estrutura de representação de conhecimento nos ambientes de desenvolvimento de sistemas baseados em conhecimento [INTELLICORP 86, 87], [LAURENT 87], [ROCHE 89], [WAH 89]. A linha de investigação seguida nesta nova geração de sistemas está baseada na construção de estruturas de dados genéricas e cuja expressividade limita-se às necessidades existentes nas aplicações reais.

Nas subseções seguintes são apresentados alguns conceitos básicos a respeito do formalismo e sistemas que os utilizam.

2.3.1 Conceitos Básicos

Um *frame* pode representar um objeto físico ou abstrato, um conceito, uma situação, etc. Estas entidades denominadas, daqui para frente "objetos", podem ser descritas por um conjunto de informações que as caracterizam. Por exemplo, num restaurante é esperado encontrar-se informações sobre o tipo de comida servida, o cardápio, o preço de cada prato e o preço total da refeição.

Um objeto é caracterizado por um conjunto de atributos denominados *slots*, aos quais estão associados valores. Alguns dos *slots* não possuem valores pré-determinados. Nestes casos existem informações ,chamadas facetas, que definem os valores admissíveis a fim de garantir a consistência do *frame*.

Semanticamente existem dois tipos de *frames*: os que representam classes de objetos e os que representam objetos individuais. Para fazer uma divisão clara destes conceitos, as classes de objetos são chamadas de **frames** e os objetos individuais de **instâncias.** Será adotada esta terminologia daqui para frente. As informações definidas nos *frames* são válidas para todas as instâncias ligadas a estes *frames*. Os *frames* podem ser especializados de modo a representarem subclasses de objetos. Por exemplo, a classe dos "CARROS" pode ser dividida nas subclasses "CARROS_0_KM" e "CARROS_USADOS". Nestes últimos *frames* só são descritas as informações que os tornam diferentes da sua superclasse "CARROS". Todas as informações descritas na superclasse são válidas nas subclasses e são acessíveis por herança. A construção de **taxonomias** de *frames*, proporcionam uma forte estruturação do conhecimento.

Instâncias podem ser pré-definidas ou ser criadas dinamicamente. Neste caso, o processo de criação de uma instância de um *frame* determinado (**instanciação**), consiste em preencher todos os *slots* definidos no *frame* em questão. A instância é criada se todos os

seus *slots* forem preenchidos corretamete, isto é, de forma consistente com as informações descritas nas facetas.

O preenchimento ou alteração do valor de um *slot* de uma instância faz com que procedimentos ligados ao *slot* em questão sejam executados automaticamente. Estes procedimentos chamados de **demons (procedural attachment)** reagem como reflexos.

A riqueza de uma linguagem de representação baseada em *frames* está ligada aos tipos de facetas que podem ser associados aos *slots*. Alguns sistemas permitem que o programador defina suas próprias facetas.

As inferências básicas possíveis estão ligadas principalmente ao processo de instanciação que, indiretamente, ativa os mecanismos de herança e execução de *demons*. Assim, a ênfase dos sistemas baseados em *frames* é dada pela sua capacidade de representar informações complexas e mantê-las consistentes(1).

A combinação de *slots*, facetas e *demons* produz uma estrutura de representação que mescla informações declarativas e procedurais, ambas colocadas de forma explícita.

2.3.2 Sistemas Baseados em Frames

Nesta subseção são apresentados três sistemas baseados em *frames*. Eles foram escolhidos por representarem formas distintas de utilização deste formalismo. Os sistemas descritos são o KL-ONE, o CENTAUR e o KEE.

Sistema KL-ONE

O sistema KL-ONE é utilizado para a representação do conhecimento em sistemas de IA. As primitivas que formam a linguagem de representação permitem a codificação de informações gerais. Estas primitivas possuem uma interpretação semântica muito bem definida, fato que caracteriza e distingue este sistema dos demais.

(1) A consistência está ligada a uma especificação coerente por parte do programador, não sendo realizada pelo sistema em si.

A tarefa de representação do conhecimento é dividida em duas partes: a descritiva, onde são definidas todas as informações conhecidas sem a preocupação de como elas podem ser usadas; e a assertiva, onde são definidos contextos para a utilização dessas informações.

Na linguagem descritiva podem ser definidos "conceitos primitivos" representando objetos que são auto definidos e "conceitos definidos" formados por conceitos primitivos e outros conceitos já definidos. Como resultado, cria-se uma rede de conceitos onde a hierarquia é uma consequência das relações entre os conceitos e não uma imposição da linguagem.

Cada conceito é definido por atributos (chamados *roles*) e por descrições das relações entre estes atributos. A riqueza semântica de KL-ONE está ligada diretamente à variedade de primitivas de relacionamento entre os atributos.

Podem ser definidos conceitos genéricos representanto informações válidas a um conjunto de objetos e conceitos individuais representando um único objeto.

A linguagem assertiva permite a manipulação dos conceitos sem que haja a necessidade de se conhecer como estão armazenados e também simplifica a interface entre o sistema e o usuário uma vez que as estruturas são muitas vezes complexas.

A principal consequência do rigor semântico com que os conceitos são criados é a possibilidade de construir-se um algoritmo de classificação capaz de determinar a posição de um conceito dentro da rede de conceitos existente a partir da comparação da estrutura dos atributos e dos conceitos.

Por fim, pode-se dizer que o sistema KL-ONE representa uma investigação com profundo rigor semântico sobre a natureza das informações existentes numa abordagem de representação centrada na noção de objeto.

Sistema CENTAUR

No sistema CENTAUR os *frames* são usados com o objetivo específico de representar informações que caracterizam doenças pulmonares. A combinação deles com regras de produção resultam numa estrutura chamada "protótipo".

A estrutura dos *frames* está adaptada ao sistema sendo, portanto, inapropriada para representar outros tipos de informações.

Num protótipo, os *slots* representam informações que necessitam ser conhecidas para poder caracterizar a doença pulmonar que representa. Estas informações são obtidas através de resultados de exames clínicos ou por inferência sobre um conjunto de regras de produção. As facetas determinam o conjunto de valores aceitáveis de uma determinada informação para que esta confirme uma das características da doença em questão.

A taxonomia de *frames* representa uma hierarquia de doenças onde, a cada nível, são descritas doenças mais específicas. O processo de utilização desta taxonomia é, naturalmente, a classificação.

Em cada protótipo estão associados *slots* de controle que contêm informações sobre o que deve ser feito caso a doença hipotética seja confirmada (via instanciação do protótipo) ou rejeitada. Este tipo de informação é necessário uma vez que existem interrelações entre as doenças e pelo fato de que a detecção (ou não) de uma certa doença possa ter efeito sobre o teste das demais.

O sistema CENTAUR representa o caso típico onde o *frame* está adaptado à um domínio de conhecimento particular de modo a representar as informações estritamente necessárias e dispor de mecanismos de inferência específicos (e portanto eficientes) às necessidades existentes.

Os *slots* de controle, associados aos protótipos, permitem que as ações de controle sejam específicas e sensíveis ao contexto. A introdução de informações de controle a nível das estruturas de conhecimento é no entanto contrária à tendência geral de isolar completamente o conhecimento do controle. Porém, neste sistema tais informações são

codificadas de forma declarativa permitindo uma reformulação simples das estratégias de controle.

Sistema KEE

O sistema KEE é um ambiente de desenvolvimento de sistemas baseados em conhecimento. Ele possui como estruturas básicas para a representação do conhecimento os *frames* e regras de produção.

Os *frames*, chamados de *units*, podem representar informações de qualquer domínio de conhecimento e são de dois tipos: o primeiro representa classes de objetos e segundo representa indivíduos.

Os *units* podem ser organizados em taxonomias através das ligações *class* , que liga um objeto individual a uma classe de objetos e *superclass* que liga duas classes de objetos. As taxonomias permitem herança múltipla, ou seja, um *unit* pode estar ligado a mais de uma superclasse.

Os *slots* dos *units* contêm valores que caracterizam o atributo do objeto, facetas que restringem os valores possíveis de serem atribuídos a eles e uma espécie de *demon*, que é ativado quando o valor de um *slot* é recuperado ou alterado. Além disso, alguns *slots* são usados para representar "métodos" implementando um tipo de programação orientada a objetos.

A manipulação dos *units* é feita através de uma linguagem especial baseada no cálculo dos predicados. Através dela, valores dos *slots* podem ser recuperados ou alterados.

No contexto do ambiente KEE, os *units* são usados como uma base de dados. A resolução de problemas propriamente dita é realizada por inferência sobre um conjunto de regras de produção. Nestas regras, as condições podem representar informações que devem ser obtidas nos *units*. Desta forma, o papel principal dos *units* no ambiente é representar objetos de modo consistente. A sua participação como componente de raciocínio é explorada em parte pelos *demons* a eles associado e, principalmente, pelas facilidades de programação orientada a objetos. Contudo, a sua combinação com as regras

de produção (sustentada por um poderoso mecanismo de manutenção da verdade (*truth-maintenance system*) chamado ATMS [deKLEER 86]) proporciona um excelente contexto para a representação e resolução de problemas.

2.4 Conclusão

Neste capítulo foi descrito o papel do conhecimento nos sistemas baseados em conhecimento, especialmente nos Sistemas Especialistas. O grau de inteligência de um Sistema Especialista está diretamente ligado à qualidade das informações nele representadas e na efetiva utilização destas.

A viabilidade dos Sistemas Especialistas passa pelo uso de formalismos de representação do conhecimento que permite definir bases de conhecimentos explícitas e declarativas. Vários formalismos já foram propostos. Dentre eles, estão os *frames*, um formalismo que permite representar informações através da noção de objetos.

Foram descritos brevemente três sistemas baseados em *frames*. Em cada um, a semântica, a estrutura e forma de utilização dos *frames* é diferente. Esta característica se, por um lado, evidencia a falta de uma teoria consensual sobre o que é realmente um *frame*, por outro, demonstra a variedade de semânticas e formas de utilização que podem ser obtidas através de uma representação centrada na noção de objeto.

Capítulo 3

A LINGUAGEM LOF

3.1 Introdução

Neste capítulo é apresentada a linguagem LOF, tema central desta dissertação. LOF está baseada no formalismo de *frames* sendo, portanto, apropriada para o desenvolvimento de Sistemas Baseados em Conhecimento.

LOF permite implementar objetos do mundo real através das noções de classes de objetos e objetos individuais representados, respectivamente, por **frames** e **instâncias.**

A apresentação está dividida em três partes. Na primeira, é mostrado o ambiente de programação em que LOF está inserida. Na segunda, são mostradas as estruturas de representação do conhecimento (*frames* e instâncias). Na terceira, completando o capítulo, são mostrados os vários mecanismos de inferência existentes.

3.2 Ambiente de Programação

A linguagem LOF, implementada em Le_Lisp [CHAILLOUX 85] sobre computador compatível com IBM PC, foi desenvolvida de modo a aproveitar o ambiente de programação fornecido por Le_Lisp. Portanto, para o desenvolvimento conveniente de bons sistemas em LOF é preciso conhecer algumas características de Le_Lisp e as facilidades que este ambiente proporciona.

Serão apresentadas nesta seção apenas algumas características gerais da linguagem Le_Lisp e do seu ambiente de programação. Para uma melhor compreensão sobre o estilo de programação LISP e sobre a linguagem Le_Lisp recomenda-se, respectivamente, a leitura de [WINSTON 89] e [CHAILLOUX 85].

A linguagem Le_Lisp tem como principal característica a facilidade de manipular expressões simbólicas. Estas expressões são amplamente utilizadas em sistemas de Inteligência Artificial pois permitem uma representação simples e direta de informações. O desenvolvimento de programas é feito de forma incremental através da definição de funções.

O ambiente de programação interativo de Le_Lisp permite que cada função seja testada logo após a sua definição diminuindo a ocorrência de erros de implementação. Esta característica facilita o rápido desenvolvimento de protótipos tornando mais interativo (e flexível) o ciclo de desenvolvimento de *software*.

Devido ao longo tempo de existência da linguagem LISP foi possível desenvolver excelentes editores de programas (orientados à sua sintaxe) e poderosos ambientes de depuração.

A linguagem LOF é composta por um conjunto de funções pré-definidas e por dois tipos de dados (os *frames* e as instâncias) usados para a representação do conhecimento. A definição de *frames* é feita de maneira análoga a definição de funções em Le_Lisp. Assim, objetos podem ser rapidamente modelados e testados.

Um *frame* é composto por informações declarativas e procedurais. As informações declarativas seguem uma sintaxe própria (descrita no apêndice 1). Já as informações procedurais são declaradas através de expressões avaliáveis pelo interpretador Le_Lisp. Desta forma, todos os recursos existentes em Le_Lisp são naturalmente acessados a partir de um *frame*.

No momento da definição de um *frame*, é feita a verificação sintáxica da sua parte declarativa. Erros sintáticos de origem procedural só serão detectados quando da sua avaliação pelo interpretador Le_Lisp, causando a interrupção da avaliação corrente e retornando o controle ao ambiente do interpretador Le_Lisp.

Um *frame* é identificado por um símbolo e não pode ser alterado após ter sido definido. Qualquer redefinição implicará na eliminação da definição anterior. Instâncias são representadas através de estruturas de dados próprias que estão descritas na seção 3.4.

Na apresentação de LOF feita no restante deste capítulo, vários exemplos, implementáveis diretamente no interpretador Le_Lisp, são apresentados. O caracter "?" é o *prompt* do interpretador Le_Lisp que espera a entrada de uma expressão para ser avaliada (como mostrado no exemplo abaixo). O símbolo "=" precede o resultado da avaliação da última expressão avaliada no fluxo de entrada.

```
? (+
? 1
?
?
? 2
?
? )
= 3
```

3.3 Frames

O *frame* é a estrutura de dados básica usada para representar o conhecimento envolvido no sistema desenvolvido. Um *frame* é composto de vários tipos de informações esquematizados na figura 3.1.

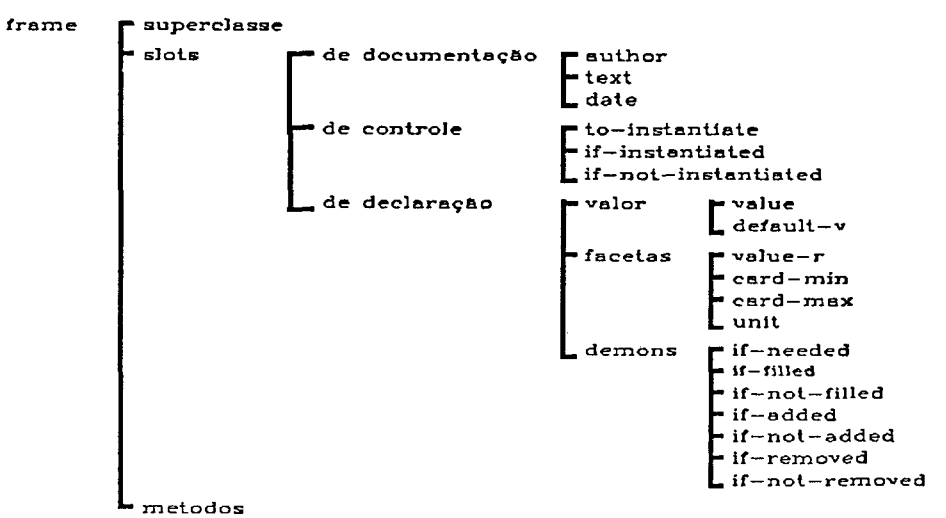


Figura 3.1 : Componentes de um *frame*

Como ilustra a figura 3.1, um *frame* LOF é formado por três tipos de informações: a superclasse, os *slots* e os métodos. A superclasse define qual é o *frame* pai do *frame* na taxonomia. Os *slots* definem os atributos do objeto representado pelo *frame*. Os métodos são procedimentos associados aos *frames* que são executados por invocação explícita de forma análoga aos métodos existentes nas Linguagens de Programação Orientada a Objetos(2).

Um *frame* é declarado através da função *deframe*. Esta função tem a seguinte sintaxe:

```
(deframe <nome>
        [(subclass-of <superclasse>)]
        [<slots>] )
```

O argumento <nome> é um símbolo que identifica o *frame*. O argumento <superclasse> identifica a superclasse do *frame* declarado. O argumento <*slots*> contém todas as definições de *slots*.

A seguir são apresentados em detalhes os componentes de representação mostrados na figura 3.1.

3.3.1 A Superclasse

Através da declaração da superclasse é possível construir taxonomias de *frames* resultando num poderoso mecanismo de inferência: a **herança**. Através deste mecanismo, toda informação que for definida em um *frame* será válida para os *frames* que são seus descendentes na taxonomia.

A sintaxe desta declaração é a seguinte:

```
(subclass-of <nomeFrame>)
```

(2) Os conceitos sobre métodos e o uso de LOF como LPOO são apresentados no capítulo 4. No presente capítulo estas informações estão omitidas propositalmente.

O argumento <nomeFrame> é um símbolo indicando o *frame* que é superclasse do *frame* declarado. Por exemplo, a declaração de que o *frame* "F2" é subclasse do *frame* "F1" é mostrada abaixo.

```
? (deframe F2 (subclass-of F1))
= F2
```

3.3.2 Slots

Os *slots* definem os atributos do objeto representado pelo *frame*. Em LOF estão definidos três tipos de *slots*: os de documentação, os de controle e os de declaração. O objetivo e sintaxe de cada tipo de *slot* são descritos a seguir.

3.3.2.1 Slots de Documentação

Os *slots* de documentação permitem, como o próprio nome indica, documentar um *frame*. A documentação visa facilitar a reutilização do software e aumentar a legibilidade da linguagem. Estão predefinidos três *slots* de documentação: *text*, *author* e *date*. Não é possível criar outros *slots*.

Os *slots* de documentação são os únicos que possuem um valor que pode ser obtido diretamente do *frame*. O valor desses *slots* não atua em nenhum processo de inferência.

text

O *slot text* contém um texto explicativo sobre o *frame*. Sua sintaxe é a seguinte:

```
(text <string>)
```

O argumento <string>, que representa o valor do *slot*, é uma cadeia de caracteres. Por exemplo, a declaração de que o *frame* "F1" é apenas um exemplo de utilização do *slot text* é mostrada abaixo.

```
? (deframe F1 (text "Exemplo de uso do slot text"))
= F1
```

A função *text-slot*, cujo parâmetro é o nome do *frame*, retorna o valor do *slot text*. Por exemplo, aplicando esta função para o *frame* "F1" acima tem-se a seguinte resposta.

```
? (text-slot 'F1)
= Exemplo de uso do slot text
```

date

O *slot date* descreve a data de criação do *frame*. Sua sintaxe é a seguinte:

```
(date (<dia> <mes> <ano>))
```

O argumento deste *slot*, que representa o seu valor, é uma lista composta de três elementos representando, respectivamente, o dia, o mês e o ano. Cada elemento pode ser de qualquer tipo válido em Le_Lisp. Por exemplo, a declaração de que o *frame* "F1" foi criado no dia 12 de fevereiro de 1990 é declarada como mostrado abaixo.

```
? (deframe F1 (date (12 fevereiro 1990)))
= F1
```

A função *date-slot*, cujo parâmetro é o nome do *frame*, retorna o valor do *slot date*. Por exemplo, aplicando esta função para o *frame* "F1" acima tem-se a seguinte resposta.

```
? (date-slot 'F1)
= (12 fevereiro 1990)
```

author

O *slot author* contém o nome da pessoa que criou o *frame*. Sua sintaxe é a seguinte:

(author <string>)

O argumento <string>, que representa o valor do *slot*, é uma cadeia de caracteres. Por exemplo, a declaração de que o *frame* "F1" foi criado pelo Leandro é mostrada abaixo.

```
? (deframe F1 (author "Leandro"))
= Leandro
```

A função *author-slot*, cujo parâmetro é o nome do *frame*, retorna o valor do *slot author*. Por exemplo, aplicando esta função para o *frame* "F1" acima tem-se a seguinte resposta.

```
? (author-slot 'F1)
= Leandro
```

3.3.2.2 Slots de Declaração

Os *slots* de declaração são definidos pelo programador e representam atributos comuns a todos os objetos pertencentes à classe modelada pelo *frame*. O poder de expressividade e de inferência de LOF vai resultar da riqueza de representação ao nível destes *slots*.

A composição dos *slots* de declaração é bastante diferente da composição dos demais *slots* e combinam informações declarativas e procedurais. Um *slot* de declaração pode conter três tipos diferentes de informações: seu valor (quando este é conhecido a priori ou por *default*), um conjunto de facetas descrevendo os valores assumíveis pelo *slot* e os *demons* que definem o comportamento das instâncias do *frame* quando da manipulação do conteúdo do *slot*.

3.3.2.2.1 Valor do Slot

A declaração do valor de um *slot* é feita quando qualquer instância do *frame* em questão deve assumir somente este valor ou o assume na ausência de informação contrária.

Cada *slot* de declaração pode conter um ou mais valores. O valor de um *slot* pode ser qualquer objeto LOF (objetos Le_Lisp ou instâncias). Quando um *slot* assume mais de um valor a representação destes valores é feita usando o tipo especial *Set*. Este tipo tem a seguinte sintaxe:

#{ <valor1> <valor2> ... <valorN>}

O tipo *Set* permite que se evite ambiguidades na representação do valor de um *slot*. Por exemplo, a representação de que um *slot* possui os valores 1, 2 e 3 deve ser feita da forma #{1 2 3}. A forma (1 2 3) representa o fato de que o *slot* contém apenas um valor, no caso, uma lista com três elementos.

A manipulação deste tipo é feita através de funções específicas (ver apêndice 2).

value

O componente *value* é utilizado quando o valor de um *slot* é conhecido a priori e não pode assumir outro valor. A sintaxe deste termo é a seguinte:

(value <valor>)

onde o argumento <valor> é uma expressão Le_Lisp. O valor do *slot* será o resultado da avaliação desta expressão.

Por exemplo, para o *frame* "TERRA" o *slot* "número-de-luas" só aceita o valor 1. Este exemplo é mostrado abaixo.

```
? (deframe TERRA (slot numero-de-luas (value 1)))
= TERRA
```

default-v

O valor por *default* de um *slot* é assumido enquanto nenhuma informação o contradisser. Esta informação permite representar conhecimento de senso comum, isto é, informações verdadeiras na maioria das situações, embora existam exceções.

Este tipo de informação está diretamente ligado às idéias de Minsky [MINSKY 75]. De acordo com Minsky um *frame* contém informações que tipicamente caracterizam um objeto e, caso não haja nenhuma informação contrária, estas informações são assumidas como sendo verdadeiras mesmo que não tenham sido explicitamente verificadas.

Em LOF, uma informação *default* só é considerada se:

1) não houver valor definido para o *slot*;
2) não houver um valor que possa ser herdado;
3) não houver uma informação contrária fornecida quando da criação de uma instância;
4) o *demon if-needed* do *slot* em questão não estiver definido.

A sintaxe desta declaração é a seguinte:

```
(default-v <valor>)
```

onde o argumento <valor> é uma expressão Le_Lisp. O valor do *slot* será o resultado da avaliação desta expressão.

Por exemplo, para o *slot* "número-portas" do *frame* CARROS o valor por *default* pode ser 2, já que é a situação mais comum (embora existam carros com 4 portas). Esta informação é declarada abaixo.

```
? (deframe CARROS (slot numero-portas (default-v 2)))
= CARROS
```

3.3.2.2.2 Facetas

As facetas associadas a um *slot* particular descrevem condições de admissibilidade para o seu preenchimento. O principal objetivo delas é garantir a integridade semântica das instâncias mantendo sempre os *slots* com valores consistentes. Após criada a instância, nenhuma alteração sobre o valor de um *slot* da instância é feita sem que o novo valor seja testado pelas facetas.

Em LOF, as características definidas nas facetas incluem a unidade em que o valor está representado, o conjunto de valores semanticamente válidos e as cardinalidades mínima e máxima de valores admissíveis ao mesmo tempo. Estas facetas estão definidas a seguir.

value-r

A faceta *value-r* (acrônimo de *value-restriction*) determina os valores admissíveis para o preenchimento do *slot*. Estes valores são definidos através de um conjunto de expressões Le_Lisp. Um valor é aceito quando o resultado da avaliação de todas as expressões for diferente de "()". As expressões mais comumente usadas são aquelas que limitam o conjunto de valores válidos (restringindo a um determinado tipo válido em Le_Lisp e/ou a um conjunto de valores) ou relações de dependência entre valores de *slot*.

No exemplo a seguir, os símbolos iniciados por "^" representam o valor do *slot* no momento da avaliação. A representação da frase "A idade dos humanos é um número

inteiro que varia de zero à 140" em LOF é representada pelo *frame* "HUMANOS" mostrado abaixo.

```
? (deframe HUMANOS
        (slot idade (value-r (integer ^idade)
        (member-of ^idade [0 140]))))
= HUMANOS
```

card-min

A faceta *card-min* define o número mínimo de valores que um *slot* deve possuir. Quando não estiver definida é assumido o valor 1. A sintaxe desta faceta é a seguinte:

(card-min <número>)

onde o argumento <número> é um número inteiro maior que zero.

Por exemplo, seja o *frame* "COMPUTADORES" com seus *slots* de declaração representando a sua configuração. Supondo que o *slot* "drives" necessite de dois valores distintos esta informação pode ser então representada pela faceta *card-min* como mostrado abaixo.

```
? (deframe COMPUTADORES
        (slot drives (card-min 2)))
= COMPUTADORES
```

card-max

A faceta *card-max* define o número máximo de valores que um *slot* deve possuir. Quando não estiver definida é assumido o valor definido na faceta *card-min*. A sintaxe desta faceta é a seguinte:

(card-max <número>)

onde o argumento <número> é um número inteiro maior que zero. Caso não exista um limite máximo o argumento <número> deve ser o símbolo especial "inf" (acrônimo de infinito).

Por exemplo, seja o *frame* "Reunião" onde está definido o *slot* "participantes". Supondo que a reunião deva ter no mínimo 3 e no máximo 10 participantes, tal informação é representada como mostrado abaixo.

```
? (deframe Reuniao
        (slot participantes (card-min 3) (card-max 10)))
= Reuniao
```

unit

A faceta *unit* é usada para representar a unidade de grandeza representada. Ela tem a única função de auxiliar o preenchimento do *slot* (ver função *read-value* no apêndice 2). Sua sintaxe é a seguinte:

```
(unit <unidade>)
```

onde <unidade> é uma cadeia de caracteres.

Por exemplo, a informação de que o *slot* "capacidade" do *frame* "Disquete" tem seu valor em k-bytes (Kb) é mostrada abaixo.

```
? (deframe Disquete (slot capacidade (unit "Kb")))
= Disquete
```

3.3.2.2.3 Demons

Os *demons* representam informações associadas com eventos ligados à manipulação dos valores dos *slots* de uma instância. Estes eventos são o preenchimento do *slot* e a sua alteração. Portanto, eles introduzem um estilo de programação orientado a dados [ROCHE 89].

O conhecimento nos *demons* é representado de forma procedural através de um conjunto de expressões Le_Lisp. Sua invocação age como um reflexo às necessidades do momento (depende do tipo de manipulação ao qual o *slot* está sujeito).

Os *demons* definidos em LOF são apresentados a seguir.

if-needed

O *demon if-needed* contém informações sobre o modo de preenchimento do *slot* quando este estiver indefinido. As expressões que o compõe são avaliadas sequencialmente pelo interpretador Le_Lisp e o valor do *slot* será o valor retornado pela última expressão avaliada. Caso este valor seja "()", por definição, o *slot* não é preenchido.

Sua sintaxe é a seguinte:

(if-needed <exp1> <exp2> ... <expN>)

onde os argumentos <expi> são expressões Le_Lisp.

Por exemplo, seja o *frame* "Quadrado" composto pelos *slots* "lado" e "área". O preenchimento do *slot* "lado" é feito através de uma pergunta ao usuário. Por sua vez, o preenchimento do *slot* "área" resulta do preenchimento do anterior e seu valor é determinado pela simples elevação ao quadrado de ^lado. A declaração deste *frame* é mostrada abaixo.

```
? (deframe Quadrado
        (slot lado
            (if-needed (print "Quanto vale o lado ?")
                (read)))
        (slot area
            (if-needed (* ^lado ^lado))))
= Quadrado
```

if-filled e if-not-filled

Os *demons if-filled* e *if-not-filled* são procedimentos executados cada vez que o valor de um *slot* for alterado respectivamente com sucesso (incluindo aqui a primeira vez que é preenchido) e com falha.

Estes *demons* são ativados quando o preenchimento do *slot* é realizado durante o processo de criação de uma instância ou pela função de manipulação de instância *put-value* (ver apêndice 2).

As funções *retry* e *confirm* (ver apêndice 2) dentro do corpo do *demon if-not-filled* possibilitam que o processo de preenchimento do *slot* seja repetido. É possível assim previnir-se do preenchimento com um valor acidentalmente errôneo (por exemplo resultante de um erro de digitação).

A sintaxe destes *demons* é a seguinte:

```
(if-filled <exp1> <exp2> ... <expN>)
(if-not-filled <exp1> <exp2> ... <expN>)
```

onde os argumentos <expi> são expressões Le_Lisp.

Por exemplo, o *frame* "Quadrado" definido acima pode ser redefinido de modo a incluir as seguintes características:

- o *slot* "lado" sempre deve ser preenchido com um número. Caso isto não aconteça deve ser apresentado uma mensagem ao usuário indicando a ilegalidade do valor fornecido e uma nova tentativa deve ser realizada;
- a cada alteração do valor do *slot* "lado" o valor do *slot* "área" deve ser atualizado.

A função *put-value*, utilizada neste exemplo, contém três argumentos: o novo valor do *slot*, o identificador do *slot* e o nome da instância. A expressão *(itself)* dentro do corpo dessa função significa que a função deve atuar na instância correntemente utilizada.

Esta nova definição do *frame* "Quadrado" é mostrada a seguir.

```
? (deframe Quadrado
        (slot lado
            (if-needed (print "Quanto vale o lado ?")
                  (read))
            (if-filled (put-value (* ^lado ^lado)
                                              'area
                                              (itself)))
            (if-not-filled (print "Valor ilegal !!")
                                        (retry)))
        (slot area
            (if-needed (* ^lado ^lado))))
= Quadrado
```

if-added e if-not-added

Os *demons if-added* e *if-not-added* são procedimentos ativados quando da adição, respectivamente, com sucesso e falha de um novo valor a um *slot*. Estes *demons* são necessários já que um *slot* pode assumir mais de um valor (conforme o valor das suas facetas de cardinalidade) e que a operação de adição de um valor não elimina os valores anteriores.

Estes *demons* reagem quando da utilização da função *add-value* (ver apêndice 2).

A sintaxe destes *demons* é a seguinte:

```
(if-added <exp1> <exp2> ... <expN>)
(if-not-added <exp1> <exp2> ... <expN>)
```

onde os argumentos <expi> são expressões Le_Lisp.

Por exemplo, seja o *frame* "Armazem" contendo o *slot* "estoque" que pode conter no máximo 5 valores representando 5 produtos quaisquer. O *demon if-added* é utilizado para enviar uma mensagem indicando que foi adicionado outro produto ao armazém e o *demon if-not-added* é utilizado para enviar uma mensagem indicando que o produto não pode ser armazenado porque o armazém já está lotado. Estas informações são declaradas no *frame* "Armazem" mostrado a seguir.

```
? (deframe Armazem
        (slot estoque
            (card-max 5)
            (if-added (print "Armazenado mais um produto!"))
            (if-not-added (print "Armazem lotado!")))
= Armazem
```

if-removed e if-not-removed

Os *demons if-removed* e *if-not-removed* são procedimentos ativados quando da remoção, respectivamente, com sucesso e falha de um valor de um *slot*. Eles são necessários já que os *slots* podem assumir mais de um valor.

Estes *demons* reagem quando da utilização da função *remove-value* (ver apêndice 2).

A sintaxe destes *demons* é a seguinte:

```
(if-removed <exp1> <exp2> ... <expN>)
(if-not-removed <exp1> <exp2> ... <expN>)
```

onde os argumentos <expi> são expressões Le_Lisp.

Por exemplo, seja o *frame* "Armazem" definido anteriormente. Supondo agora que, por questão de mercado, decida-se que o estoque nunca deva ser inferior a dois produtos. Esta informação é representada através da faceta *card-min* que assume o valor 2. Usando os *demons if-removed* e *if-not-removed* pode-se produzir, de forma análoga aos *demons* de

adição, mensagens indicando o resultado da operação de remoção de um produto do armazém. Estas informações são declaradas no *frame* "Armazem" definido abaixo.

```
? (deframe Armazem
        (slot estoque
            (default-v #{produto-1 produto2})
            (card-min 2)
            (card-max 5)
            (if-removed (print "Removido um produto!"))
            (if-not-removed (print "Atingido limite minimo !"))))
= Armazem
```

3.3.2.3 Slots de Controle

O principal processo de inferência em LOF é a instanciação. Criar uma instância de um determinado *frame* consiste em preencher todos os seus *slots* de declaração. Os *slots* de controle permitem estabelecer condições para que se possa iniciar este preenchimento e definir os efeitos colaterais associados à criação (ou não) de uma instância.

Em LOF estão definidos os seguintes *slots* de controle: *to-instantiate*, *if-instantiated* e *if-not-instantiated*. A cada *slot* de controle está associado um valor. Este valor é composto por um conjunto de expressões Le_Lisp. Estas expressões são avaliadas sequencialmente produzindo os efeitos necessários.

Os *slots* de controle de LOF são descritos a seguir.

to-instantiate

As expressões no corpo deste *slot* são avaliadas no início do processo de instanciação, antes do preenchimento dos *slots* de declaração. Para que a instanciação prossiga o resultado da avaliação de todas as expressões deve ser diferente de ().

A sintaxe deste *slot* é a seguinte:

(to-instantiate <exp1> ... <expN>)

onde cada <expi> é uma expressão Le_Lisp.

if-instantiated

As expressões no corpo deste *slot* são avaliadas quando uma instância do *frame* em questão for criada, ou seja, após o preenchimento de todos os *slots* de declaração.

A sintaxe para declaração deste *slot* é a seguinte:

(if-instantiated <exp1> ... <expN>)

onde cada termo <expi> é uma expressão Le_Lisp.

if-not-instantiated

As expressões no corpo deste *slot* são avaliadas quando a tentativa de criação de uma instância falhar, isto é, quando pelo menos um *slot* de declaração não for preenchido.

A sintaxe para declaração deste *slot* é a seguinte:

(if-not-instantiated <exp1> ... <expN>)

onde cada termo <expi> é uma expressão Le_Lisp.

3.4 Instâncias

As instâncias são estruturas de dados representando objetos individuais. Estes objetos são definidos diretamente pelo programador ou, mais comumente, gerados por instanciação de um *frame* particular.

Em LOF uma instância sempre pertence à classe de objetos associada ao *frame* que a gerou. Sua estrutura é formada pelos *slots* de declaração definidos no *frame* associado e seus respectivos valores. Além disso contém um identificador do *frame* que a gerou e do tipo da instância (detalhado a seguir).

A estrutura de dados que usada para representar uma instância tem a seguinte sintaxe:

#<<tipo> <classe> <slot$_1$> <valor$_1$> ... <slot$_N$> <valor$_N$>>

onde <tipo> é um símbolo que identifica o tipo da instância (*frame* ou *classif*), <classe> é um símbolo identificando o *frame* que gerou a instância, <slot$_i$> são símbolos identificadores de *slots* de declaração e <valor$_i$> os seus respectivos valores.

As instâncias geradas por instanciação de um *frame* particular contêm informações consistentes com as especificações estruturais definidas nas facetas dos *slots* do *frame* em questão. Esta consistência é mantida mesmo quando da alteração dos valores dos *slots*, feitas através de funções LOF de manipulação de instâncias.

Por exemplo, seja o *frame* "CARROS_ZERO" composto pelos *slots* "ano", "portas" e "km-rodados".

```
? (deframe CARROS_ZERO
        (slot ano (default-v 1990))
        (slot portas (default-v 2)
            (value-r (member-of ^portas {2 4})))
        (slot km-rodados (value 0)))
= CARROS_ZERO
```

Uma instância do *frame* "CARROS_ZERO" é gerada através da sua instanciação (usando-se a função *make*) como mostrado a seguir.

```
? (setq carro-1 (make 'CARROS_ZERO))
= #<frame CARROS_ZERO ano 1990 portas 2 km-rodados 0>
```

Alterações dos valores dos *slots* só são aceitas se forem consistentes com as facetas correspontentes. Por exemplo, o *slot* "portas" rejeitará qualquer valor diferente de 2 ou 4 e o *slot* "km-rodados" não aceita nenhuma alteração do seu valor.

```
? (put-value 3 'portas carro-1)
= ()
? carro-1
= #<frame CARROS_ZERO ano 1990 portas 2 km-rodados 0>
? (put-value 4 'portas carro-1)
= t
= #<frame CARROS_ZERO ano 1990 portas 4 km-rodados 0>
? (put-value 4 'km-rodados carro-1)
= ()
? carro-1
= #<frame CARROS_ZERO ano 1990 portas 4 km-rodados 0>
```

Em LOF estão definidos dois tipos de instâncias. Cada tipo possui características peculiares ao processo de criação que os tornam apropriados para usos diferentes. As instâncias do tipo *frame* são usadas na instanciação direta de um *frame* (vista no exemplo anterior). As instâncias do tipo *classif* resultam de uma classificação *top-down* numa taxonomia particular. Estes dois tipos são descritos a seguir.

3.4.1 As instâncias do tipo frame

Na semântica original dos *frames*, os *slots* representam informações que caracterizam um objeto e, portanto, para que uma instância represente tal objeto todos os *slots* devem ser preenchidos com valores válidos.

Na linguagem LOF, as instâncias do tipo *frame* respeitam esta semântica. Isto significa que para gerar instâncias deste tipo todos os *slots* de declaração do *frame* em questão devem ser preenchidos.

As instâncias do tipo *frame* são geradas por avaliação da função *make*. Esta função retorna a instância criada cujo tipo é identificado pelo símbolo *frame* na primeira posição

da sua estrutura de representação da instância. Caso algum *slot* não possa ser preenchido a instância não é criada e a função *make* retorna ().

3.4.2 As instâncias do tipo classif

A classificação é um importante mecanismo de inferência em LOF (ver seção 3.5.5). Ela gera, como resultado, um conjunto de instâncias de *frames* da taxonomia. Estas instâncias, na sua essência, têm a mesma semântica das instâncias do tipo *frame*, ou seja, resultam da criação de objetos individuais consistentes com as facetas dos *slots* do *frame* associado. Porém, alguns mecanismos na sua criação são diferentes, especialmente a herança. Assim, para que as funções LOF possam tratar adequadamente as instâncias estas são identificadas pelo símbolo *classif* na primeira posição de sua estrutura de representação.

As instâncias do tipo *classif* resultam da avaliação da função *classify* que desencadeia o processo de classificação *top-down* a partir de um *frame* passado como parâmetro da chamada. Um *frame* gera somente uma instância a cada classificação e esta é acessada através do símbolo que identifica o *frame* que a gerou.

Por exemplo, uma classificação no *frame* "CARROS_ZERO" produz a seguinte situação.

```
? (classify 'CARROS_ZERO)
= t
? CARROS_ZERO
= #<classif CARROS_ZERO ano 1990 portas 4 km-rodados 0>
```

3.5 Mecanismos de Inferência

A linguagem LOF possui vários mecanismos de inferência associados aos *frames* de modo a permitir a manipulação de todos os tipos de informações neles representadas.

Em LOF, inferências podem ser realizadas por herança, quando definido a priori ou por *default* o valor do *slot*, por procedimento, por instanciação e classificação. Estes mecanismos estão descritos a seguir.

3.5.1 A Herança

Os *frames* podem ser organizados em taxonomia, isto é, manterem relações hierárquicas entre si. Esta relação permite definir que um *frame* "F2" seja uma subclasse da classe representada pelo *frame* "F1". A herança faz com que toda informação representada no *frame* "F1" seja válida para o *frame* "F2". Em LOF, um *frame* herda apenas os *slots* de controle e de declaração (excluindo portanto os *slots* de documentação).

Os *slots* de declaração herdam os seus componentes (valores, *demons* e facetas). Por exemplo, o processo de preenchimento de um *slot* pode ser igual numa taxonomia mas os valores aceitáveis diferentes, segundo a localização do *frame* nesta (quanto mais profundo mais restritivo o conjunto de valores aceitáveis).

Como exemplo, sejam os *frames* "ALUNO", "ALUNO_REPROVADO" e "ALUNO_APROVADO" declarados a seguir.

```
? (deframe ALUNO
        (slot nome
            (if-needed (read-value "Nome do aluno :")))
        (slot nota
            (value-r (member-of ^nota [0 10]))
            (if-needed (read-value "Qual a sua nota :"))))

= ALUNO
?
```

```
? (deframe ALUNO_REPROVADO
        (subclass-of ALUNO)
        (slot nota
            (value-r (member-of ^nota [0 5[))
        (if-instantiated (print "O aluno " ^nome
                                    " foi reprovado")))
= ALUNO_REPROVADO
?
? (deframe ALUNO_APROVADO
        (subclass-of ALUNO)
        (slot nota
            (value-r (member-of ^nota [5 10]))
            (if-filled (print "O aluno " ^nome
                                    " foi aprovado."))))
= ALUNO_APROVADO
```

Graças ao mecanismo de herança o *demon if-needed* do *slot* "nota" não precisa ser redefinido nos *frames* "ALUNO_APROVADO" e "ALUNO_REPROVADO".

3.5.2 Valor do Slot

Os componentes *value* e *default-v* são usados quando o valor do *slot* é conhecido respectivamente a priori ou por *default*. Assim em ambos os casos, a inferência permite determinar o valor de um *slot* durante a criação de uma instância.

O componente *value* indica que o *slot* correspondente somente aceita um valor determinado a priori. Esta informação é prioritária sobre qualquer outra para o preenchimento do *slot* em questão.

O componente *default-v* representa informações de senso comum, ou seja, informações que são verdadeiras para a maioria dos casos, embora existam exceções. Em LOF, um valor de *slot* por *default* só é utilizado para preenchê-lo quando não houver nenhuma outra informação a seu respeito (ou seja, quando o termo *value* ou o *demon if-needed* não forem definidos; quando este valor não puder ser herdado de outra instância; ou quando não for definido no contexto da instanciação).

3.5.3 Inferência por Procedimento

As inferências procedurais estão associadas aos *demons* e a faceta *value-r*. Os *demons* permitem inferir o valor de um *slot* e executar ações associadas ao seu preenchimento. A faceta *value-r* (*value-restriction*) estabelece condições sobre o valor preenchido.

A potencialidade deste tipo de inferência está na variedade de condições que podem ser produzidas (desde uma simples pergunta ao usuário até a avaliação de funções Le_Lisp complexas).

3.5.4 A Instanciação

O processo de instanciação, que é o principal mecanismo de inferência em LOF, pode ser visto como sendo um casamento (*matching*) entre uma observação (composta por *slots* e seus respectivos valores) com um modelo de objeto representado pelo *frame*. O casamento é válido quando todos os *slots* de declaração do *frame* forem preenchidos.

Como apresentado na seção 3.4, estão definidos em LOF dois tipos de instâncias: o tipo *frame* e o tipo *classif*. O processo de criação associado a cada tipo apresenta particularidades. A seguir é apresentado o algoritmo de criação de instância do tipo *frame*. Para o outro tipo só será apresentada as modificações deste algoritmo básico.

3.5.4.1 Algoritmo de Instanciação

O algoritmo de criação de instâncias do tipo *frame* é composto de três etapas:

- o teste do *slot to-instantiate*;
- o preenchimento dos *slots* de declaração;
- a avaliação dos efeitos resultantes da instanciação.

1 - Primeira Etapa

A primeira etapa do algoritmo consiste em testar as condições definidas no *slot* de controle *to-instantiate*. Caso este *slot* não esteja definido então a primeira etapa é encerrada com sucesso e o fluxo de controle é passado para a segunda etapa. Caso contrário todas as expressões Le_Lisp definidas no corpo do *slot* são avaliadas sequencialmente. Se o resultado da avaliação de uma das expressões for igual a "()" então a etapa é encerrada como fracasso e o fluxo de controle do algoritmo é desviado para a terceira etapa. Senão, a primeira etapa é encerrada como sucesso e o fluxo é passado para a segunda etapa.

Eventualmente alguma condição definida no *slot to-instantiate* requer o valor de um *slot* de declaração. Neste caso o *slot* deve ser preenchido na forma indicada na segunda etapa. Caso ele não possa ser preenchido considera-se que a condição não foi satisfeita e, portanto, a primeira etapa é encerrada com fracasso e o fluxo de controle é desviado para a terceira etapa.

2 - Segunda Etapa

A segunda etapa consiste em preencher todos os *slots* de declaração definidos no *frame* em questão e aqueles herdados dos *frames* ascendentes na taxonomia.

O preenchimento de um *slot* segue os seguintes passos:

1) a determinação do valor do *slot*;

2) a validação do valor obtido no passo anterior;

3) a avaliação dos efeitos resultantes do preenchimento (ou não) do *slot*.

Estes três passos são executados para todos os *slots* a serem preenchidos. Caso algum não possa ser preenchido então esta segunda etapa é encerrada com fracasso. Caso contrário ela é encerrada com sucesso. Em ambas as situações o fluxo de controle é

passado para a terceira etapa. Os três passos descritos acima são agora apresentados em detalhes:

Passo 1

O preenchimento de um *slot* pode ser feito de uma das seguintes formas:

i) se o valor for fornecido no contexto da instanciação (isto é, como parâmetro da função de instânciação *make* (ver apêndice 2) no caso de instâncias do tipo *frame*) então o *slot* será preenchido por este valor;

ii) se o valor for descrito no componente value associado ao *slot* então este será preenchido pelo valor resultante da avaliação da expressão definida naquele componente. Caso esteja definido, a avaliação nunca poderá retornar "()". Neste caso ocorrerá um erro e o processo de instanciação será interrompido;

iii) se o valor for descrito no *demon if-needed* associado ao *slot* então este será preenchido pelo valor da última avaliação feita no corpo daquele *demon*. O *slot* só será considerado preenchido caso o valor que lhe for atribuído for diferente de "()". Caso contrário o algoritmo considera que o *slot* não pode ser preenchido e a forma seguinte (iv) não é considerada;

iv) se o valor for descrito no componente *default-v* associado ao *slot* então este será preenchido pelo valor resultante da avaliação da expressão definida naquele componente. Caso este componente seja utilizado sua avaliação nunca poderá retornar "()". Neste caso ocorrerá um erro e o processo de instanciação será interrompido.

Caso não seja possível determinar o valor do *slot* por nenhuma das quatro formas acima então é considerado que o *slot* não pode ser preenchido e o fluxo de controle é desviado para o passo 3. Caso contrário o fluxo de controle segue para o passo 2.

Passo 2

No passo 2 a consistência do valor preenchido é verificada. Este passo não é efetuado caso o *slot* tenha sido preenchido nos itens ii ou iv do passo 1. Esta verificação de consistência envolve as facetas de cardinalidade (*card-min* e *card-max*) e a faceta *value-r*. A ordem dos testes é a seguinte:

i) a cardinalidade do *slot* deve ser maior ou igual ao valor definido na faceta *card-min*;

ii) a cardinalidade do *slot* deve ser menor ou igual ao valor definido na faceta *card-max*;

iii) a avaliação de todas as expressões definidas na faceta *value-r* devem retornar valores diferentes de "()".

Caso uma destas três restrições não for satisfeita, então o algoritmo considera que o *slot* não foi preenchido com um valor consistente. Caso contrário, o valor é aceito. Em ambas as situações o fluxo de controle segue para o passo 3.

Passo 3

No passo 3 os *demons* associados ao preenchimento do *slot* são ativados. O algoritmo não executa este passo caso o *slot* tenha sido preenchido nos itens ii ou iv do passo 1. Caso o *slot* tenha sido preenchido com um valor consistente então o *demon if-filled* será executado automaticamente. Caso contrário o *demon if-not-filled* será executado automaticamente.

Se em ambos os casos os respectivos *demons* não estiverem definidos este passo torna-se sem efeito.

Terceira Etapa

A terceira etapa consiste em avaliar os efeitos ligados à criação ou não da instância. Caso a primeira e segunda etapas tenham sido encerradas com sucesso então a instância é criada e, como efeito colateral, as expressões do *slot* de controle *if-instantiated* são avaliadas. Caso contrário a instância não é criada e, como efeito colateral, as expressões do *slot* de controle *if-not-instantiated* são avaliadas.

Se em ambas as situações os respectivos *slots* de controle não estiverem definidos esta etapa torna-se sem efeito.

3.5.4.2 Dependência Entre Slots

Em vários pontos do algoritmo descrito acima expressões Le_Lisp são avaliadas. Estes pontos compreendem as avaliações dos *demons if-needed*, *if-filled* e *if-not-filled*, da faceta *value-r* e dos *slots* de controle *to-instantiate*, *if-instantiated* e *if-not-instantiated*.

Na prática relações de dependência entre os valores dos *slots* de declaração podem existir. Por exemplo, para o *frame* "Quadrado" o valor do *slot* "área" depende do valor do *slot* "lado" como mostrado a seguir.

```
? (deframe Quadrado
        (slot area
            (if-needed (* ^lado ^lado)))
        (slot lado
            (if-needed (read-value "Valor do lado: "))))
= Quadrado
```

Durante a avaliação, pelo interpretador Le_Lisp, da expressão "(* ^lado ^lado)" no corpo do *demon if-needed* do *slot* "area", é preciso conhecer o valor do *slot* "lado". Neste momento, o sistema LOF "suspende" a avaliação desta expressão e inicia o preenchimento do *slot* "lado" segundo os três passos definidos na segunda etapa do algoritmo de instanciação. Caso o *slot* seja preenchido a expressão é efetivamente avaliada retornando o valor do *slot* "area". Porém, caso não tenha sido possível preencher o *slot* "lado" a avaliação

da expressão é interrompida sem causar um erro de interpretação. Como, neste exemplo, a expressão fazia parte do *demon if-needed* o passo 1 da segunda etapa do algoritmo é encerrado com fracasso, causando o não preenchimento do *slot* "area", implicando na não criação da instância.

Caso seja detectado uma dependência cíclica entre os *slots* o processamento é interrompido, causando um erro de execução ao nível do interpretador Le_Lisp.

A habilidade do algoritmo de instanciação em tratar a dependência entre *slots* proporciona uma excelente expressividade à linguagem LOF.

3.5.4.3 Instâncias do tipo frame

Instâncias do tipo *frame* são criadas pela função *make* (ver apêndice 2). O algoritmo é aquele apresentado na seção 3.5.4.1. Um exemplo de criação de uma instância do tipo *frame* é dado no exemplo 3 do apêndice 3.

3.5.5 Classificação

A semântica original atribuída aos *frames* define que eles representam estereótipos de situações cujas características são representadas pelos *slots* com suas respectivas informações [GOMEZ 81]. Neste sentido, criar uma instância significa testar se a situação representada pelo *frame* é verificada ou não num contexto particular.

O mecanismo de classificação consiste em criar uma instância para cada *frame* da taxonomia, partindo-se do *frame* raiz até atingir as folhas da taxonomia. A passagem de nível na taxonomia só é feita se o *frame* do nível superior for instanciado. Caso contrário todos os *frames* abaixo dele são ignorados (poda).

A classificação é muito utilizada nos sistemas de diagnósticos médicos (como CENTAUR [AIKINS 83] e MDX [GOMEZ 81]). Nestes sistemas, os dados iniciais são informações a respeito do paciente. Tais informações, acrescidas de outras solicitadas pelo

sistema durante a criação de instâncias, definem o estado do paciente e caracterizam uma ou mais doenças.

3.5.5.1 Instâncias do tipo classif

As instâncias de classificação são geradas durante o processo de classificação. O algoritmo de criação destas instâncias é análogo ao apresentado na seção 3.5.4.1. com as seguintes alterações:

- a classificação sempre deve iniciar pelo *frame* raiz de uma taxonomia;
- na segunda etapa, devem ser preenchidos somente os *slots* de declaração definidos no *frame* envolvido na instanciação (sem considerar os *slots* herdados);
- no passo 1 da segunda etapa, é acrescida uma outra forma de preenchimento do *slot*: a herança de valor de uma instância de classificação acima na taxonomia. Este processo é prioritário sobre o item i do algoritmo. Na instância criada o *slot* assim preenchido não está sendo descrito já que isto levaria a uma duplicação de informação.

À instância criada é associado um símbolo cujo valor é igual ao nome do *frame* que a gerou. Isto permite a sua identificação para posterior consultas dos valores dos *slots*.

3.6 Conclusão

Neste capítulo foi apresentada a linguagem LOF. Esta linguagem está baseada num formalismo de representação de conhecimento centrado em *frames* e pode ser utilizada para o desenvolvimento de Sistemas Especialistas.

A linguagem está inserida no ambiente de programação da linguagem Le_Lisp (um dialeto LISP). Assim, as características de interatividade, facilidade de manipulação simbólica e rápida prototipagem existentes nesta linguagem podem ser aproveitadas no

desenvolvimento de sistemas em LOF. Qualquer função definida em Le_Lisp é naturalmente executada a partir de um programa LOF.

A resolução de problemas a partir de LOF é feita através da implementação de *frames* e instâncias que representam, respectivamente, classes de objetos e objetos individuais. Vários mecanismos de inferência são disponíveis e, dentre eles, o principal é o de instanciação.

Um *frame* contém informações acerca da estrutura e do comportamento de uma classe de objetos. As instâncias, geradas a partir de um *frame* particular, contém informações consistentes com as especificações (de estrurua e comportamento) nele definidas. Graças ao mecanismo de classificação é possível implementar Sistemas Especialistas de diagnóstico. Nestes sistemas, a taxonomia de *frames* representa uma árvore de diagnósticos possíveis. Estes são confirmados ou rejeitados usando-se o mecanismo de instanciação.

Capítulo 4

LOF E PROGRAMAÇÃO ORIENTADA A OBJETO

4.1 Introdução

Neste capítulo é mostrado como a linguagem LOF pode ser usada como uma Linguagem de Programação Orientada a Objetos (LPOO).

Após serem apresentadas as características funcionais do paradigma de Programação Orientada a Objetos (POO) é feito um paralelo com o modelo de Programação Orientada a *Frames* (POF) e é mostrado de que forma estas características podem ser implementadas em LOF. Um novo tipo de instância é então introduzido, o objeto. São então descritas as modificações trazidas ao algoritmo básico de criação de instâncias para que suporte este novo tipo. São descritos também os métodos, como sendo o mecanismo que determina o comportamento do novo tipo de instância. Um exemplo de POO em LOF é dado como ilustração. Finalmente, conclui-se sobre a expressividade de um paradigma híbrido incluindo POF e POO.

4.2 Elementos de Programação

A programação orientada a objetos está baseada em três conceitos: os objetos, as mensagens e as classes [RAMAMOORTHY 88]. Um objeto representa, através de um conjunto de variáveis locais, o estado de uma entidade. Estas variáveis não são acessíveis diretamente. O acesso para leitura ou modificação de uma variável local de um objeto é realizado através do envio de uma mensagem ao objeto. Cada objeto está associado a uma classe. Na classe estão definidos os atributos que os objetos desta classe possuirão e os métodos relacionados às mensagens que poderá receber, definindo o comportamento dos objetos.

Os *frames* de LOF e os objetos da POO apresentam características estruturais semelhantes. Os *frames* são formados por *slots* e os objetos por variáveis internas cuja semântica é idêntica aos *slots*. Além disso, tanto os *frames* como os objetos são organizados em hierarquias visando o compartilhamento de informações comuns que são acessadas por herança. A diferença principal entre os dois formalismos está ligada aos seus comportamentos.

Na POO, não existem condições associadas à criação de uma instância como ocorre em LOF e as variáveis internas normalmente são indefinidas no momento da sua criação. O comportamento de uma dada instância é definido pelo conjunto de métodos associados à classe que a gerou. Somente através da invocação explícita de métodos tem-se acesso para consulta ou alteração aos valores das variáveis locais. Já em LOF, uma instância somente é criada se todos os *slots* foram preenchidos com valores válidos e em nenhum momento um *slot* pode estar indefinido. O seu comportamento é definido essencialmente pela ação reflexa dos *demons*.

Baseando-se no parágrafo anterior, conclui-se que o *frame* apresenta duas diferenças fundamentais em relação ao objeto definido nas LPOO. Primeiro, o *frame* é incapaz de representar informações indefinidas. Segundo, o comportamento de uma instância é determinado automaticamente pelos *demons* associados aos *slots* e não pela "vontade" de quem a utiliza.

4.3 LOF como Linguagem de Programação Orientada a Objetos

Para permitir que LOF possa ser usada como uma LPOO, foi criado mais um tipo de instância, chamado *object* e a estrutura do *frame* foi ampliada de modo que possam ser definidos métodos. Estas alterações são descritas em detalhes nas duas subseções seguintes.

4.3.1 Instâncias do tipo object

As instâncias do tipo *object*, identificadas pelo símbolo *object* na primeira posição da estrutura de representação associada aos objetos individuais e seus *slots* armazenam valores que representam o estado corrente dos objetos.

A principal diferença com os demais tipos de instâncias é que nem todos os *slots* precisam ter valor definido e alguns *slots* podem até passar a terem valor indefinido (representado por convenção por "()"). Porém, os *slots* preenchidos contêm valores consistentes com as facetas associadas. Caso seja atribuído um valor inconsistente à um *slot* indefinido este será rejeitado, fazendo com que o *slot* permaneça indefinido.

Instâncias do tipo *object* são criadas pela função *new* (ver apêndice 2). O algoritmo de instanciação é composto somente pela segunda etapa do algoritmo de base descrito na seção 3.5.4.1 e com as seguintes modificações:

- nem todos os *slots* de declaração precisam ser preenchidos para que a instância possa ser criada. Os que não forem preenchidos ficarão indefinidos (representado pela lista vazia "()");
- sempre a função *new* criará uma instância (mesmo que todos os *slots* fiquem com valores indefinidos);
- no passo 1, um *slot* só pode ser preenchido nos itens i, ii e iv. Se o *slot* só contiver o *demon if-needed* (correspondendo ao item iii) o seu valor será indefinido;
- no passo 2, caso o valor do *slot* não seja consistente então este valor é descartado e o valor fica indefinido.

No caso de haver dependência entre *slots*, as mesmas alterações descritas acima são aplicáveis para o preenchimento dos *slots* dependentes, com exceção do fato do *slot* não poder ser determinado através do *demon if-needed*.

Por exemplo, seja o *frame* "Homem" caracterizado pelos *slots* "sexo", cujo valor é "masculino" e "nome" cujo valor deve ser uma cadeia de caracteres. Este *frame* é definido como segue.

```
? (deframe Homem
        (slot sexo (value 'masculino))
        (slot nome (value-r (stringp ^nome))))
= Homem
```

Instâncias do tipo *object* conterão, logo após a sua criação, o *slot* "nome" com valor indefinido.

```
? (setq homem-1 (new 'Homem))
= #<object Homem sexo masculino nome ()>
? (put-value "Fulano de Tal" 'nome homem-1)
= t
? homem-1
= #<object Homem sexo masculino nome Fulano de Tal>
? (put-value 5 'nome homem-1)
= ()
? homem-1
= #<object Homem sexo masculino nome Fulano de Tal>
? (clear-slot 'nome homem-1)
= t
? homem-1
= #<object Homem sexo masculino nome ()>
? (put-value 'Um_Simbolo 'nome homem-1)
= ()
? homem-1
= #<object Homem sexo masculino nome ()>
```

4.3.2 Os Métodos

Um método é uma função Le_Lisp particular definida pela função de declaração de métodos *demethod*. Assim como o *slot*, ele é herdado entre os *frames* de uma taxonomia. Um método pode ser utilizado em qualquer tipo de instância.

A sintaxe de declaração de um método é a seguinte:

(demethod <symbol> <lvar> <s1> ... <sN>)

onde <symbol> identifica o nome do método, <lvar> é uma lista contendo os argumentos do método e os <s_i> são expressões Le_Lisp que compõem o corpo do método.

Por exemplo, seja o *frame* "Conta_Bancaria" contendo o *slot* "saldo". Duas operações comuns a este tipo de objeto são o **depósito** e o **saque** de dinheiro. Tais operações são definidas pelos métodos "deposito" e "saque" como é ilustrado na declaração a seguir.

```
? (deframe Conta_Bancaria
        (slot saldo (default-v 0))
            (demethod deposito (valor)
                    (put-value (+ valor ^saldo) 'saldo (itself))
                    (print "Saldo alterado para : "
                                (+ valor ^saldo)))
                    t
            (demethod saque (valor)
                    (if (< (- ^saldo valor) 0)
                            (and (print "Falta de fundos
                               para este saque !")
                               ())
                               (put-value (- ^saldo valor)
                                       'saldo
                                       (itself))
                               (print "Saldo restante : " ^saldo)
                               t)))
= Conta_Bancaria
```

A expressão *(itself)* que aparece como parâmetro da função *put-value* em ambos os métodos significa que a instância que está sofrendo a alteração de "saldo" é a passada como parâmetro da função que avalia o método.

A avaliação de um método, correspondendo ao envio de uma mensagem nas LPOO, é feita através da função *call-method*. Esta função retorna o valor da última avaliação feita no corpo do método.

```
? (setq minha-conta (object 'Conta_Bancaria))
= #<object Conta_Bancaria saldo 0>
? (call-method 'deposito minha-conta 500)
Saldo alterado para: 500
= t
? minha-conta
= #<object Conta_Bancaria saldo 500>
? (call-method 'saque minha-conta 450)
Saldo restante: 50
= t
? minha-conta
= #<object Conta_Bancaria saldo 50>
? (call-method 'saque minh-conta 100)
Falta de fundos para este saque !
= ()
? minha-conta
= #<object Conta_Bancaria saldo 50>
```

4.4 Exemplo: Criação do Objeto Pilha

Neste exemplo é definido uma pilha do tipo LIFO onde duas operações básicas podem ser realizadas: **empilhar** e **desempilhar** elementos. Por outro lado, funções anexas podem ser implementadas sobre este tipo de objeto como o teste do estado da pilha (vazia ou cheia), a leitura do elemento no topo da pilha e sua visualização na tela. A representação das capacidades de armazenamento da pilha pode ser feita definindo-se o seu tamanho.

Baseado nas especificações acima, é mostrado a seguir a a declaração dos *frames* "Pilha" e "Pilha_Limitada" representando os dois tipos de pilha. O *slot* "conteudo" contém uma lista que representa os dados empilhados. A representação de que a pilha está vazia no momento da sua criação é dada pelo valor *default* "(vazia)". (Não é possível representar uma pilha vazia através de () pois isto é interpretado pelo sistema como sendo uma indefinição do valor do *slot* e, consequentemente, as expressões Le_Lisp que necessitam o valor do *slot* não são avaliadas). No *frame* "Pilha_Limitada", o *slot* "tamanho" contém um número que indica a limitação da pilha.

```
? (deframe Pilha
        (slot conteudo (default-v '(vazia)))

        (demethod empilha (valor)
            (put-value2 (cons valor ^conteudo)
                        'conteudo
                        (itself))
            (call-method 'mostra (itself))
            valor)

        (demethod desempilha ()
            (if (call-method 'esta_vazia (itself))
                        "erro : pilha vazia"
                        (let ((v (car ^conteudo)))
                            (put-value2 (cdr ^conteudo)
                                        'conteudo
                                        (itself))
                            (call-method 'mostra (itself))
                            v)))

        (demethod esta_vazia ()
                        (and (equal ^conteudo '(vazia)) t))

        (demethod retorna_topo ()
            (if (call-method 'esta_vazia (itself))
                        "erro : pilha vazia"
                        (car ^conteudo)))

        (demethod mostra ()
            (if (call-method 'esta_vazia (itself))
                    (and (print "[]") t)
                    (mapcar '(lambda (x) (print "[ " x " ]"))
                                    ^conteudo)
                     t)))
= Pilha
?
? (deframe Pilha_Limitada
        (subclass-of Pilha)

        (slot tamanho
            (if-needed (read-value "Tamanho da pilha : ")))

        (demethod empilha (valor)
            (if (call-method 'esta_cheia (itself))
                    "erro : pilha cheia"
                    (put-value2 (cons valor ^conteudo)
                                    'conteudo
                                    (itself))
                    (call-method 'mostra (itself))
                    valor))
```

```
        (demethod esta_cheia ()
            (and (equal?^tamanho (1- (length ^conteudo))) t)))

= Pilha_Limitada
```

Exemplos de utilização destes tipos são dados pelos diálogos a seguir.

```
? (setq p1 (new 'Pilha))
= #<object Pilha conteudo (vazia)>
? (call-method 'mostra p1)
[]
= t
? (call-method 'empilha p1 10)
[ 10 ]
= 10
? p1
= #<object Pilha conteudo (10 vazia)>
? (call-method 'empilha p1 20)
[ 20 ]
[ 10 ]
= 20
? p1
= #<object Pilha conteudo (20 10 vazia)>
? (call-method 'retorna_topo p1)
= 20
? (call-method 'esta_vazia p1)
= ()
? (call-method 'desempilha p1)
[ 10 ]
= 20
? (call-method 'mostra p1)
[ 10 ]
= t
? (call-method 'desempilha p1)
[]
= 10
? p1
= #<object Pilha conteudo (vazia)>
? (call-method 'desempilha p1)
= erro : pilha vazia
? (call-method 'retorna_topo p1)
= erro : pilha vazia
```

```
? (setq p1 (new 'Pilha_Limitada))
= #<object Pilha_Limitada  conteudo (vazia) tamanho ()>
? (call-method 'empilha p1 55)
Qual o tamanho da pilha : 2
[ 55 ]
= 55
? p1
= #<object Pilha_Limitada conteudo (55 vazia) tamanho 2>
? (call-method 'empilha p1 100)
[ 100 ]
[ 55 ]
= 100
? p1
= #<object Pilha_Limitada conteudo (100 55 vazia) tamanho 2>
? (call-method 'empilha p1 200)
= erro : pilha cheia
? p1
= #<object Pilha_Limitada conteudo (100 55 vazia) tamanho 2>
? (call-method 'desempilha p1)
[ 55 ]
= 100
? p1
= #<object Pilha_Limitada conteudo (55 vazia) tamanho 2>
```

4.5 Conclusão

Neste capítulo foi apresentado a utilização da linguagem LOF como Linguagem de Programação Orientada a Objetos. Através dos *slots* de definição, dos métodos e das instâncias do tipo *object* LOF permite que o programador disponha, além do paradigma de programação orientado a *frames*, o paradigma de programação orientado a objetos.

A utilização tradicional dos *frames* (via instanciação e classificação) representa um poderoso mecanismo de processamento onde as instâncias criadas têm como papel principal representar objetos ou situações. Pode-se dizer, grosso modo, que no paradigma de programação orientado a *frames* o processamento termina com a criação de uma instância. Os *demons* e facetas associados aos *slots* têm como objetivo principal garantir a consistência das instâncias criadas. No paradigma de Programação Orientada a Objetos, o

processamento começa com a criação das instâncias. Elas participam ativamente (através da troca de mensagens) da dinâmica do sistema.

A linguagem LOF permite a combinação natural destes dois paradigmas de programação, flexibilizando a utilização dos *frames*.

Capítulo 5

EXEMPLO DE UTILIZAÇÃO DE LOF : UM SISTEMA ESPECIALISTA PARA DIAGNÓSTICO DE DEFEITOS EM EQUIPAMENTOS DE SUBESTAÇÕES DE ALTA TENSÃO

5.1 Introdução

Neste capítulo é apresentado um exemplo do uso de LOF como gerador de Sistema Especialista. O sistema desenvolvido é um sistema de diagnóstico de defeito de equipamentos numa Subestação de Alta Tensão tirado de [LACERDA 87] e objetiva mostrar o poder de expressão de LOF na resolução de problemas reais.

Na segunda seção, características de uma Subestação de Alta Tensão são apresentadas destacando-se alguns dos equipamentos que a compõe e descrevendo o diagnóstico destes quando apresentam defeitos. Na terceira seção é mostrada a organização do Sistema Especialista através da descrição dos módulos de conhecimento que o compõe. A apresentação de cada módulo contém a descrição e formas de utilização dos *frames* que o compõem.

5.2 Subestações de Alta Tensão

Uma Subestação de Alta Tensão é um nó de um Sistema Elétrico com as seguintes funções [LACERDA 87]:

- alterar a topologia do Sistema Elétrico;
- transformar as suas tensões de entrada para níveis adequados à transmissão e distribuição de energia elétrica;
- monitorar as grandezas elétricas que lhe são pertinentes;

- proteger as instalações e linhas quando da ocorrência de condições anormais de funcionamento.

Para atingir tais objetivos é necessário um Sistema de Controle na subestação.

Uma das funções do Sistema de Controle é a supervisão Ela "permite determinar a situação das linhas que saem e/ou entram da subestação e dos seus equipamentos internos, [...] emitindo alarmes quando ocorrerem defeitos, impedindo manobras que possam comprometer o sistema e/ou equipamentos (intertravamentos) e coordenando, quando necessário, o fluxo de informações entre vários elementos integrantes da subestação. A supervisão deve ser também capaz de detetar a necessidade de execução de serviços de manutenção, sejam estes de caráter corretivo ou preventivo." [LACERDA 87].

5.2.1 Os Equipamentos de Uma Subestação

A descrição da estrutura de uma Subestação de Alta Tensão é feita de forma natural através da descrição dos seus equipamentos e da organização existente entre eles.

Um dos níveis de estruturação de uma Subestação de Alta Tensão é o *bay*. Os *bays* dividem a subestação em módulos que são compostos por vários equipamentos. Cada *bay* pode conter tipos e números diferentes de equipamentos. Os equipamentos tipicamente encontrados são os transformadores, os disjuntores, os relés, as chaves secionadoras, etc. Cada equipamento pode ser caracterizado por dois tipos de informações:

- um de origem estática que permite identificar e descrever o equipamento (seu fabricante, seu modelo, seu código de identificação do arranjo topológico da subestação, etc.);
- um de origem dinâmica que descreve o estado e as condições de operação do equipamento. Este tipo de informação é representado pelo estado dos contatos, sensores e de suas condições de operação.

5.2.2 Diagnóstico de Equipamentos

O diagnóstico de defeitos dos equipamentos é baseado nas informações de origem estática e dinâmica. As combinações possíveis das informações dinâmicas levam os diversos diagnósticos possíveis.

Um diagnóstico é formado por duas informações: a causa do defeito e uma recomendação sobre a manutenção que deve ser realizada.

Por exemplo, um disjuntor pode apresentar uma das seguintes condições de operação: "Não Abre", "Não Fecha", "Não Opera" ou "Normal". Os disjuntores Mitsubishi do modelo X1 possuem os seguintes contatos que supervisionam as suas condições de funcionamento: 63GA, 63AA, 63AL, 63GL, 8DC1, 8DC2, 8A, 8SA e 49M. O defeito "Nao Opera" associado aos contatos 63GA e 63GL ligados leva ao seguinte diagnóstico: "Grave vazamento de gás" e recomenda "Providenciar Manutenção Urgente".

5.3 Organização do Sistema Especialista

O Sistema Especialista descrito neste capítulo objetiva diagnosticar defeitos em equipamentos internos da subestação, sendo portanto enquadrado na função de supervisão dentro do Sistema de Controle.

O sistema desenvolvido é organizado em dois conjuntos básicos de *frames*. O primeiro conjunto contém *frames* que representam uma Subestação de Alta Tensão específica, e constituem a Base de Conhecimento (BC). O segundo conjunto contém *frames* cujo objetivo é representar informações de controle sobre a utilização do sistema. Esses *frames* constituem a Máquina de Inferência (MI) do sistema.

Para implementar a BC e a MI o sistema está organizado em quatro módulos (como mostrados na figura 5.1). Os módulos 1, 2 e 3, que formam a BC, contêm, respectivamente, os *frames* que representam a subestação e seus equipamentos, as instâncias dos *frames* do módulo 1 que representam o estado corrente de toda a subestação e os *frames* que

representam taxonomias de diagnóstico de equipamentos. O módulo 4 contém os *frames* descrevendo a MI.

Os *frames* definidos neste sistema são dados no apêndice 4.

5.3.1 Representação da Subestação

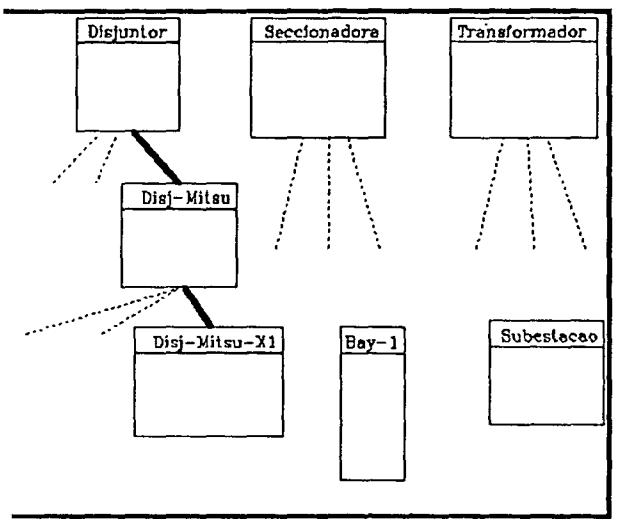
O módulo 1, que representa a subestação, contém *frames* descrevendo dois tipos de informações relativas a descrição: 1) dos equipamentos propriamente ditos e 2) da composição dos *bays* e da subestação.

A descrição da subestação é composta pelo nome da subestação e a lista de *bays* que a constitui como definido abaixo.

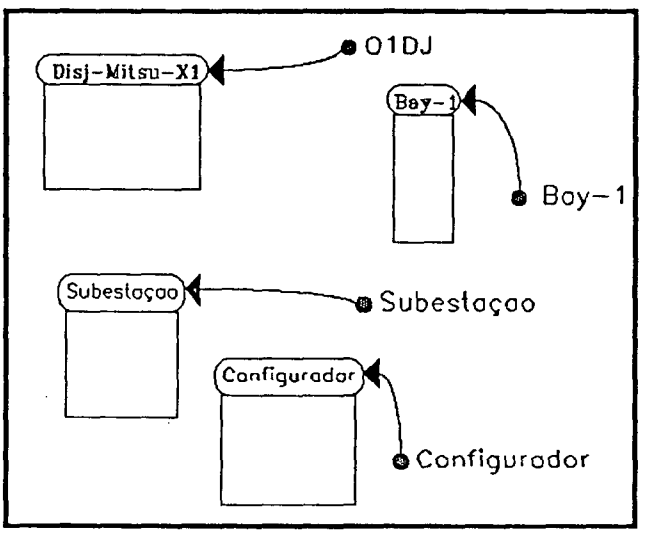
```
? (deframe Subestacao
        (slot nome (value "Substacao-1"))
        (slot bays (value '(1 2 3 4))))
= Subestacao
```

A descrição de um *bay* da subestação contém seu identificador e as informações sobre os equipamentos que o constituem . O *frame* "Bay_1" abaixo contém a declaração do *bay* 1 do sistema implementado.

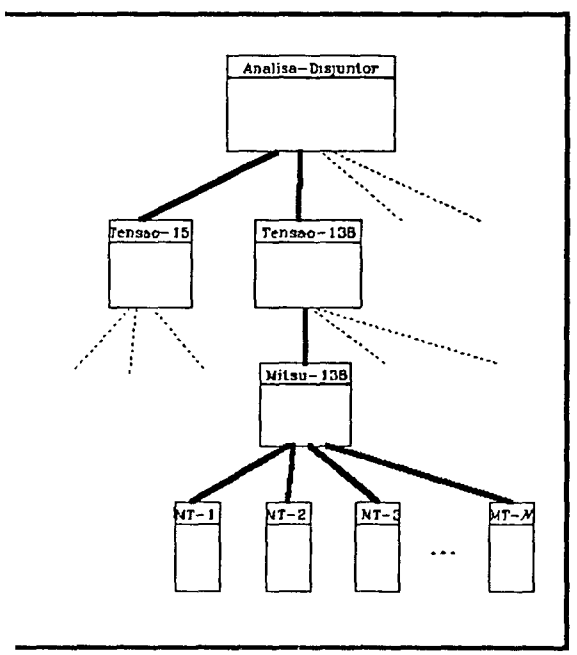
```
(deframe Bay_1
        (slot bay (value 1))
        (slot tensao (value 138))
        (slot nome (value "LT_Entrada"))
        (slot disjuntor (value '01DJ))
        (slot secionadora-1 (value '01CDA))
        (slot secionadora-2 (value '01CDB))
        (slot secionadora-3 (value '01CDC))
        (slot equipamentos (value '(disjuntor secionadora-1
        seccionadora-2 seccionadora-3))))
```



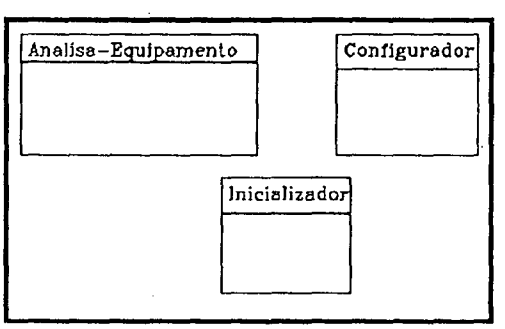
Modulo 1 : Objetos da Subestação



Modulo 2 : Base de Dados



Modulo 3 : Taxonomias Para Diagnostico



Modulo 4 : Controle

Figura 5.1 : Módulos de Conhecimento do Sistema

A descrição dos equipamentos é feita através de taxonomia específica a cada equipamento aproveitando (através da herança) a descrição hierárquica das propriedades comuns. Por exemplo, para um disjuntor estão definidos os *slots* "codigo", "fabricante", "modelo", "nome", "operacionalidade", "config"(3), "contato" (um *slot* para cada contato). A taxonomia iniciada pelo *frame* "Disjuntor" no módulo 1 da figura 5.1 descreve os vários tipos de disjuntor existentes na subestação.

5.3.2 Base de Dados

O módulo da Base de Dados tem o papel equivalente à Memória de Trabalho dos sistemas de regras de produção, ou seja, apresenta o estado corrente do problema. No Sistema Especialista em questão, a Base de Dados é formada por instâncias dos *frames* representando o estado dos equipamentos existentes na subestação.

Para permitir o acesso às informações da Base de Dados por parte de instâncias de outros módulos, as estruturas que representam as instâncias são acessadas através de identificadores globais (apontadores Le_Lisp). Assim, são definidos os identificadores "Subestação", "Bay_1", "Bay_2", etc. para descrever a subestação e seus *bays* e um identificador para cada equipamento existente. Por exemplo, para o disjuntor do *bay* 1 está definido o símbolo "01DJ" (como mostrado no módulo 2 da figura 5.1).

5.3.3 As Taxonomias de Diagnóstico de Equipamentos

Este módulo é formado por taxonomias associadas ao diagnóstico de cada tipo de equipamento. O diagnóstico de um equipamento é efetuado por classificação.

Por exemplo, a taxonomia para diagnóstico de disjuntor é iniciada pelo *frame* "Analisa_Disjuntor" mostrado no módulo 3 da figura 5.1. Nesta taxonomia, a geração de diagnósticos envolve três tipos de informação:

- a tensão de entrada do disjuntor (o segundo nível da taxonomia);

(3) O papel deste `slot` é explicado na seção 5.3.4.

- o fabricante (o terceiro nível da taxonomia);
- contatos e operacionalidade (quarto e quinto níveis da taxonomia).

5.3.4 O Controle do Sistema

O módulo 4 é formado por *frames* que coordenam a utilização do sistema. No sistema implementado, foram definidos os três *frames* seguintes neste módulo: "Inicializador", "Configurador" e "Analisa_Equipamento".

A instanciação do *frame* "Inicializador" produz, como efeito colateral ao preenchimento do seu *slot* "equipamentos", a criação das instâncias do módulo Base de Dados e do *frame* "Configurador".

O *frame* "Configurador" contém o método "configura" que permite definir estados dos equipamentos existentes na Base de Dados (usando para isso o valor do *slot* "config" definido para o equipamento).

O *frame* "Analisa_Equipamento" é usado criando uma instância deste em que é solicitado o *bay* e o equipamento com defeito. Logo a seguir é invocado o processo de classificação do equipamento escolhido usando os *frames* do módulo 3.

O princípio de utilização do SE é o seguinte. Num primeiro momento o usuário configura o estado dos equipamentos descritos na Base de Dados, simulando um defeito(4). Uma vez feita esta configuração, a classificação é ativada de modo a determinar as causas do defeito em questão, da qual resulta um diagnóstico.

(4) Num sistema real, o usuário seria substituído por sensores diretamente ligados sobre o equipamento. Estes sensores alimentariam automaticamente a base de dados.

5.4 Conclusão

Este capítulo apresentou um Sistema Especialista para diagnóstico de defeitos em equipamentos de Subestações de Alta Tensão desenvolvido na linguagem LOF.

O SE apresenta como principal característica a modularidade do conhecimento: novos equipamentos podem ser introduzidos, diagnósticos mais elaborados podem ser produzidos e novas configurações de subestações podem ser criadas com a simples inclusão de novos *frames*, sem que isto afete os *frames* já existentes.

Vários mecanismos de inferência foram usados nesta implementação. Por exemplo, nos módulos de Representação da Subestação e de Base de Dados os *frames* e instâncias foram usados como estruturas de representação do conhecimento, aproveitando conceitos das linguagens de programação orientada a objeto como, por exemplo, a hierarquia de classes de objetos e a herança associada. No módulo de diagnóstico os *frames* representam hipóteses cuja confirmação resulta da classificação. No módulo de Controle os *frames* representam roteiros de atividades a serem desenvolvidas através do preenchimento dos seus *slots*.

Embora o Sistema Especialista descrito neste capítulo tenha objetivos puramente acadêmicos, pode-se concluir, baseando-se na modularidade do sistema, que a forte estruturação do conhecimento provida pelo formalismo de *frames* aliada aos vários mecanismos de inferência existentes em LOF propiciam um modo eficaz de desenvolvimento de sistemas desta natureza.

Capítulo 6

CONCLUSÃO

Nesta dissertação foi apresentada LOF, uma linguagem de desenvolvimento de Sistemas Especialistas totalmente integrada à linguagem LISP. Esta linguagem está baseada no formalismo de representação do conhecimento centrado em *frames*.

Neste formalismo o domínio de conhecimento do sistema é mapeado na forma de objetos. Tais objetos são definidos por um conjunto de atributos chamados *slots*. Os *slots*, por sua vez, são caracterizados por facetas que permitem manter a unidade semântica do objeto. Aliado à capacidade de representação, os frames dispõem de vários mecanismos de inferência.

Outra característica dos *frames* é a forte estruturação dada ao conhecimento proporcionada pelo paradigma através da organização hierárquica dos *frames*. Através desta hierarquia forma-se o poderoso conceito de herança que faz com que informações comuns a vários *frames* sejam descritas em um único *frame* e sejam herdadas para todos os *frames* que estejam abaixo dele na taxonomia.

Em LOF, os *frames* representam estereótipos de objetos ou situações. Assim, os *frames* são usados como entidades genéricas das quais são criadas instâncias. Sob este enfoque, um *frame* representa um modelo (*pattern*) e o processo de instanciação representa o casamento (*matching*) deste modelo com uma observação composta por *slots* e seus valores.

A linguagem de definição de *frames* (e instâncias) está baseada na sintaxe de Le_Lisp, que foi a linguagem usada para a sua implementação. São combinadas expressões declarativas e procedurais. Esta combinação de estilos flexibiliza a programação uma vez que os procedimentos (expressões Le_Lisp) são geralmente pequenos e estão associados a um conhecimento declarativo cujo contexto dentro do *frame* está bem determinado.

Em LOF estão definidos os mecanismos de inferência por herança (proporcionado pela organização hierárquica dos frames), por procedimento (proporcionado pelos *demons* associados aos *slots*), por instanciação (que consiste em preencher todos os *slots* definidos para um *frame* e por classificação (que permite classificar um objeto dentro da taxonomia de *frames*. Neste mecanismo, usado para a criação de sistemas de diagnóstico, os *frames* representam os diagnósticos possíveis e os *slots* as informações necessárias para a sua confirmação).

Através da linguagem LOF o programador dispoõe de três paradigmas de programação:

- o funcional, proporcionado pela linguagem de base Le_Lisp;
- o orientado a *frames*, que é o paradigma central que consiste em criar *frames* e instâncias representando o conhecimento sobre o problema tratado pelo Sistema Especialista;
- o orientado a objetos, através da "degeneração" da estrutura de um *frame* aliada aos métodos transformando-o em um objeto no sentido clássico usado nas linguagens de programação orientadas a objetos.

A adição dos paradigmas funcional e orientado a objetos dinamiza o uso tradicional dos *frames* permitindo que eles assumam outras semânticas como, por exemplo, representar um roteiro de atividades cuja ordem é dada pela sequência de *slots* a serem preenchidos.

A versão de LOF atualmente implementada contém todas as características descritas acima. Embora a complexidade da estrutura dos *frames* prejudique a eficiência do sistema (comprometendo aplicações fortemente restritas no tempo) existe a contrapartida associada a expressividade da linguagem, de modo que objetos razoavelmente complexos sejam facilmente modelados e, por outro lado, que a base de conhecimentos formada por instâncias possa ser mantida consistente (através das facetas associadas aos *slots*).

Como perspectivas de continuação deste trabalho tem-se as seguintes propostas:

- construir um editor de *frames* orientado à sintaxe LOF;
- estudar uma metodologia de desenvolvimento de Sistemas Especialistas adequada ao formalismo de *frames*;
- estudar quais os benefícios específicos obtidos com a combinação dos três paradigmas de programação em relação à representação do conhecimento;
- estudar a viabilidade da combinação com outros formalismos de representação do conhecimento, em especial as regras de produção integrando, por exemplo, LOF à linguagem SP1 [KAESTNER 89].

APÊNDICE 1

DESCRIÇÃO SINTÁTICA DA LINGUAGEM LOF

A seguir é apresentada uma descrição sintática da linguagem LOF na forma BNF [DONOVAN 84]. Nesta BNF os símbolos não terminais serão cercados por "<" e ">" e os que estiverem sublinhados são símbolos não terminais Le_Lisp definidos como em [CHAILLOUX 85]. Também são utilizados os metasímbolos "::=" (é definido por), "*" (repetição zero ou mais vezes da expressão quantificada), "+" (repetição uma ou mais vezes da expressão quantificada), "|" (ou), "{" e "}" (que cercam elementos opcionais).

Os *frames* e instâncias que compõem um programa LOF são definidos pelos termos <frame> e <instância> abaixo.

```
<frame>          ::= (deframe <symbol> <colrpoFrame>* )

<corpoFrame>     ::= <superclasse> |
                     <método>     |
                     <slot>

<superclasse>    ::= (subclass-of <symbol> )

<método>         ::= (demethod <symbol> ( <symbol>* )
                     <sexpr>* )

<slot>           ::= <slotdeDocumentação> |
                     <slotdeControle>    |
                     <slotdeDeclaração>

<slotdeDocumentação> ::= (text <string> )  |
                     (date (<number> <symbol>
                           <number>))|
                     (author <string> )

<slotdeControle>  ::= (to-instantiate <sexpr>* )    |
                     (if-instantiated <sexpr>* )    |
                     (if-not-instantiated <sexpr>* )

<slotdeDeclaração> ::= (slot <symbol> <corpoSlot>* )

<corpoSlot>      ::= <valor> |
                     <demon> |
```

<faceta>

<valor> ::= (value <valor-slot>) |
(default-v <valor-slot>)

<facetas> ::= (unit <string>) |
(value-r <sexpr>*) |
(card-min <integer>) |
(card-max <máximo>)

<máximo> ::= inf |
<integer>

<demons> ::= (if-needed <sexpr>*) |
(if-filled <sexpr>*) |
(if-not-filled <sexpr>*) |
(if-added <sexpr>*) |
(if-not-added <sexpr>*) |
(if-removed <sexpr>*) |
(if-not-removed <sexpr>*)

<instância> ::= #< <tipo> <symbol> <slot-valor> >

<tipo> ::= frame |
object |
classif

<slot-valor> ::= <symbol> <valor-slot>

<valor-slot> ::= <instância> |
<sexpr>

APÊNDICE 2

AS FUNÇÕES LOF

A utilização da linguagem LOF dentro do ambiente de programação Le_Lisp é realizada através de um conjunto de funções relacionadas aos *frames* e às instâncias. Estas funções têm a mesma sintaxe e regras de operação que as funções Le_Lisp.

As funções LOF podem ser divididas em quatro grupos:

- as relativas aos *frames* (seção 2.1)
- as relativas às instâncias (seção 2.2)
- as relativas aos tipos computacionais *Set*, *integer* e *real* (seção 2.3)
- as funções gerais (seção 2.4)

A apresentação da sintaxe das funções segue a notação definida em [CHAILLOUX 85].

2.1 Funções Relativas aos Frames

(deframe <symb> <s1> <s2> ... <sN>) [FSUBR]

Define um novo *frame* de nome <symb>. Os argumentos <si> são expressões LOF descritas na terceira seção do capítulo 3. A função retorna <symb>.

```
? (deframe f1)
= f1
? (deframe f1)
** deframe : frame redefinido : f1
= f1
```

(all-slots <symb>) **[SUBR]**

Retorna uma lista com os *slots* de declaração definidos no *frame* de nome <symb>.

```
? (deframe f1 (slot s1) (slot s2))
= f1
? (all-slots 'f1)
= (s1 s2)
```

(ppf <symb>) **[FSUBR]**

Exibe na tela uma definição na forma identada (*pretty print*) do *frame* de nome <symb>. Esta função sempre retorna t.

```
? (deframe f1 (slot s1) (slot s2 (if-needed (read))))
= f1
? (ppf f1)
(deframe f1
        (slot s1)
        (slot s2
            (if-needed (read))))
= t
```

(framep <symb>) **[SUBR]**

Retorna <symb> se o *frame* de nome <symb> for definido. Caso contrário retorna ().

```
? (framep 'f1)
= ()
? (deframe f1)
= f1
? (framep 'f1)
= f1
```

(superclass <symb>) **[SUBR]**

Retorna o nome da superclasse do *frame* de nome <symb>. Caso o *frame* não tenha superclasse retorna ().

```
? (deframe f1)
= f1
? (superclass 'f1)
= ()
? (deframe f2 (subclass-of f1))
= f2
? (superclass 'f2)
= f1
```

(subclasses <symb>) **[SUBR]**

Retorna uma lista com as subclasses do *frame* de nome <symb>. Caso o *frame* não tenha subclasses retorna ().

```
? (deframe f1)
= f1
? (subclasses 'f1)
= ()
? (deframe f2 (subclass-of f1))
= f2
? (deframe f3 (subclass-of f1))
= f3
? (subclasses 'f1)
= (f2 f3)
```

(author-slot <symb>) **[SUBR]**

Retorna o valor do *slot* de documentação *author* do *frame* de nome <symb>. Caso o *slot* não esteja definido retorna ().

```
? (deframe f1)
= f1
? (author-slot 'f1)
= ()
? (deframe f1 (author "leandro"))
** deframe : frame redefinido : f1
= f1
? (author-slot 'f1)
= leandro
```

(date-slot <symb>) **[SUBR]**

Retorna o valor do *slot* de documentação *date* do *frame* denome <symb>. Caso o *slot* não esteja definido retorna ().

```
? (deframe f1)
= f1
? (date-slot 'f1)
= ()
? (deframe f1 (date (25 outubro 1989))
** deframe : frame redefinido : f1
= f1
? (date-slot 'f1)
= (25 outubro 1989)
```

(text-slot <symb>) **[SUBR]**

Retorna o valor do *slot* de documentação *text* do *frame* de nome <symb>. Caso o *slot* não esteja definido retorna ().

```
? (deframe f1)
= f1
? (text-slot 'f1)
= ()
? (deframe f1 (text "Um frame exemplo"))
= f1
? (text-slot 'f1)
= Um frame exemplo
```

(reject-hypo <symb>) **[SUBR]**

Impede que o *frame* de nome <symb> possa gerar uma instância do tipo *classif*. Ver item 3.5.5.

(remove-rejection <symb>) **[SUBR]**

Permite que o *frame* de nome <symb> possa gerar uma instância do tipo *classif*. Ver item 3.5.5.

(remove-all-rej) [FSUBR]

Permite que todos os *frames* possam gerar suas instâncias do tipo *classif*. Ver item 3.5.5.

(rejected) [FSUBR]

Retorna uma lista contendo todos os *frames* que estão impossibilitados de gerar instância do tipo *classif*. Ver item 3.5.5.

2.2 Funções relativas a uma Instância

(make <sym1> <sym2> <s1> ... <symN> <sN-1>) [SUBR]

Cria uma instância do tipo *frame* do frame de nome <sym1>. Retorna a instância criada ou () quando esta não pode ser criada. Os parâmetros opcionais <sym2> <s1> até <symN> <sN-1> definem um contexto para a instanciação e representam pares de atributos (*slots*) e valores conhecidos.

```
? (deframe Quadrilatero
        (slot l1
            (if-needed (read-value "Valor lado 1 ?"))
            (if-filled (print "Preenchido lado 1 !")))
        (slot l2
            (if-needed (read-value "Valor lado 2 ?"))
            (if-filled (print "Preenchido lado 2 !")))
        (slot l3
            (if-needed (read-value "Valor lado 3 ?"))
            (if-filled (print "Preenchido lado 3 !")))
        (slot l4
            (if-needed (read-value "Valor lado 4 ?"))
            (if-filled (print "Preenchido lado 4 !")))
            (if-instantiated (print "Instanciado Quadrilatero
            !")))

= Quadrilatero
?
? (deframe Quadrado
        (subclass-of Quadrilatero)
        (to-instantiate (print "Instanciando Quadrado")
            (equal ^l1 ^l2)
            (equal ^l2 ^l3)
            (equal ^l3 ^l4))
        (slot area
            (if-needed ( * ^l1 ^l2))
            (if-filled (print "Preenchido slot area !")))
        (if-instantiated (print "Instanciou Quadrado !"))
```

```
        (if-not-instantiated (print "Nao inst. Quadrado !")))
= Quadrado
?
? (make 'Quadrado 'l1 5 'l4 5)
Instanciando Quadrado
Preenchido slot l1.
Valor lado 2 ? 5
Preenchido slot l2.
Valor lado 3 ? 5
Preenchido slot l3.
Preenchido slot l4.
Preenchido slot area !
Instanciou Quadrado !
= #<frame Quadrado l1 5 l2 5 l3 5 l4 5 area 25>
? (make 'Quadrado 'l1 5)
Instanciando Quadrado.
Preenchido slot l1.
Valor lado 2 ? 10
Nao instanciou Quadrado.
= ()
```

(new <sym1> <sym2> <s1> ... <sym3> <sN-1>) **[SUBR]**

Cria uma instância do tipo *object* a partir do *frame* de nome <sym1>. Os parâmetros opcionais <sym2> <s1> até <symN> <sN-1> definem um contexto para a instanciação e representam pares de atributos (*slots*) e seus valores.

Considerando o *frame* "Quadrado" definido anteriormente tem-se o seguinte exemplo:

```
? (new 'Quadrado)
= #<object Quadrado l1 () l2 () l3 () l4 () area ()>
?
? (new 'Quadrado 'l1 5 'l2 5)
Preenchido slot l1.
Preenchido slot l2.
= #<object Quadrado l1 5 l2 5 l3 () l4 () area ()>
```

(classify <sym1> <sym2> <s1> ... <symN> <sN-1>) **[SUBR]**

Ativa o mecanismo de inferência classificação (iniciada a partir do *frame* de nome <sym1>, criando, como efeito colateral, instâncias do tipo *classif*. Esta função sempre retorna t. Os parâmetros opcionais <sym2> <s1> até <symN> <sN-1> definem um contexto para a classificaão e representam pares de atributos (*slots*) e com seus valores.

```
? (classify 'Quadrilatero)
Valor lado 1 ? 5
Preenchido slot l1.
Valor lado 2 ? 5
Preenchido slot l2.
Valor lado 3 ? 5
Preenchido slot l3.
Valor lado 4 ? 5
Preenchido slot l4.
Instanciou Quadrilatero
Instanciando Quadrado
Preenchido slot area.
Instanciou Quadrado.
= t

? Quadrilatero
= #<classif Quadrilatero l1 5 l2 5 l3 5 l4 5>

? Quadrado
= #<classif Quadrado area 25>

? (classify 'Quadrilatero 'l1 5 'l2 5 'l4 10)
Preenchido slot l1.
Preenchido slot l2.
Valor lado 3 ? 5
Preenchido slot l3.
Preenchido slot l4.
Instanciado Quadrilatero.
Instanciando Quadrado.
Nao instanciou Quadrado.
= t

? Quadrilatero
= #<classif Quadrilatero l1 5 l2 5 l3 5 l4 10>
```

Como o *frame* "Quadrado" não foi instanciado, o símbolo "Quadrado" não está definido.

(class <inst>) [SUBR]

Retorna o nome do *frame* que gerou a instância <inst>.

```
? (deframe f1
        (slot s1
            (value 10)))
= f1
? (setq a (make 'f1))
= #<frame f1 s1 10>
? (class a)
= f1
? (setq a (new 'f1))
= #<object f1 s1 10>
? (class a)
= f1
? (classify 'f1)
= t
? (class f1)
= f1
```

(instp <s>) [SUBR]

Retorna <s> se argumento for uma instância. Caso contrário retorna ().

```
? (deframe f1
        (slot s1
            (value 10)))
= f1
? (instp '(3 4 5))
= ()
? (instp 65)
= ()
? (instp "string")
= ()
? (instp #[a b c])
= ()
? (instp (new 'f1))
= #<frame f1 s1 10>
? (instp #<frame Teste slot bola>)
= #<frame Teste slot bola>
```

(instype <inst>) **[SUBR]**

Retorna o tipo da instância <inst>. Os tipos definidos são: *frame*, *object* e *classif*.

```
? (deframe f1
        (slot s1
            (value 10)))
= f1
? (setq a (make 'f1))
= #<frame f1 s1 10>
? (instype a)
= frame
? (setq a (new 'f1))
= #<object f1 s1 10>
? (instype a)
= object
? (classify 'f1)
= t
? (instype f1)
= classif
```

(get-slot <symb>) ou ^<symb> **[FSUBR]**

(get-slot <symb>/<inst>) ou ^<symb>/<inst> **[FSUBR]**

A função *get-slot* (ou o macrocaracter equivalente "^") retorna o valor do *slot* <symb> da instância <inst>. Caso o argumento <inst> esteja omitido então é assumido que a instância é a que está correntemente sendo utilizada. Caso o valor do *slot* não esteja definido (para instâncias do tipo *object*) esta função ativa o processo de preenchimento do *slot* <symb>.

```
? (deframe f1
        (slot s1
            (value-r (< ^s1 10))
            (if-needed ^s10/^s2))
        (slot s2
            (value #<object f2 s10 100>)))
= f1
```

Numa instanciação, a expressão ^s10/^s2 é avaliada em duas etapas. Primeiro é determinado o valor do *slot* "s2" (no exemplo a instância #<object f2 s10 100> e depois é determinado o valor do *slot* "s10" (no exemplo 100).

A função *get-slot* permite que a avaliação de uma expressão Le Lisp seja interrompida (sem causar erro) caso um dos argumentos da expressão seja um *slot* que não pode ter seu valor determinado.

(slot-value <symb> <inst>) **[SUBR]**

Retorna o valor do *slot* <symb> na instância <inst>. Caso este não esteja definido (em instâncias do tipo *object*) é ativado o processo de preenchimento do *slot*. Se o preenchimento não puder ser realizado a função retorna "()". A diferença desta função com a função *get-slot* é que esta não interrompe a avaliação de uma expressão Le_Lisp que tenha como argumento um *slot* cujo valor não pode ser determinado (o que possivelmente causará um erro de execução).

```
? (deframe NOTA_FISCAL
        (slot valor
            (if-needed (read-value "Valor da compra : ")))
        (slot desconto
            (default-v 0.2))
        (slot valor-final
            (if-needed (- ^valor (* ^valor ^desconto)))
            (if-filled (print "Preenchido valor final !"))))
= NOTA_FISCAL
?
? (setq nota1 (new 'NOTA_FISCAL 'valor 100))
= #<object NOTA_FISCAL valor 100 desconto 0.2 valor-final ()>
?
? (slot-value 'valor nota1)
= 100
?
? (slot-value 'desconto nota1)
= 0.2
?
? (slot-value 'valor-final nota1)
Preenchido valor final !
= 80
?
? nota1
= #<object NOTA-FISCAL valor 100 desconto 0.2 valor-final 80>
```

(clear-slot <symb> <inst>) **[SUBR]**

Torna indefinido o valor do *slot* <symb> da instância <inst> (que deve ser do tipo *object*).

```
? (deframe Quadrado
        (slot lado
            (if-needed (read-value "Lado :"))
            (if-filled (clear-slot 'area (itself))
                       (slot-value 'area (itself))))
```

```
        (slot area
            (if-needed (* ^lado ^lado))
            (if-filled (print "Preenchido slot area !"))))
= Quadrado
?
? (setq quad (new 'Quadrado 'lado 5))
Preenchido slot area !
= #<object Quadrado lado 5 area 25>
?
? (clear-slot 'area quad)
= t
?
? quad
= #<object Quadrado lado 5 area ()>
?
? (put-value 4 'lado quad)
Preenchido slot area !
= t
?
? quad
= #<object Quadrado lado 4 area 16>
```

(put-value <s> <symb> <inst>) [SUBR]

Preenche o *slot* <symb> da instância <inst> com o valor <s>. Este valor só é armazenado se passar pelas restricções definidas pelas facetas do *slot*. Caso contrário é mantido o valor anterior. Quando o *slot* é preenchido com um valor válido o *demon if-filled* é avaliado e o valor retornado da função é t. Caso contrário, é avaliado o *demon if-not-filled* e o valor retornado pela função é ().

```
? (deframe Quadrado
        (slot lado
            (if-needed (read-value "Lado : "))
            (value-r (numberp ^lado))
            (if-not-filled (print "Lado deve ser numerico !"))
            (if-filled (put-value (* ^lado ^lado)
                                  'area
                                  (itself))))
        (slot area))
= Quadrado
?
? (setq quad (make 'Quadrado))
Lado : 10
= #<frame Quadrado lado 10 area 100>
?
```

```
? (put-value 5 'lado quad)
= t
?
? quad
= #<frame Quadrado lado 5 area 25>
?
? (put-value 'valor-invalido 'lado quad)
Lado deve ser numerico !
= ()
?
? quad
= #<frame Quadrado lado 5 area 25>

? (put-value2 20 'lado quad)
= t
```

(put-value2 <s> <symb> <inst>) **[SUBR]**

Tem o mesmo objetivo que a função *put-value*. A diferença é que esta função não leva em consideração a existência de facetas ou dos *demons if-filled/if-not-filled*. Esta função é normalmente usada quando a instância tem a mesma semântica dos objetos clássicos da programação orientada a objeto.

Considerando o frame "Quadrado" definido anteriormente, tem-se o seguinte diálogo.

```
? (setq quad (make 'Quadrado))
Lado : 10
= #<frame Quadrado lado 10 area 100>
?
? quad
= #<frame Quadrado lado 20 area 25>
?
? (put-value2 'valor-invalido 'lado quad)
= t
?
? quad
= #<frame Quadrado lado valor-invalido area 25>
```

(add-value <s> <symb> <inst>) **[SUBR]**

Adiciona o valor <s> ao conjunto de valores já existentes para o *slot* <slot> na instância <inst>. O valor só será armazenado se o conjunto de valores resultante desta adição passar pelas restricções definidas pelas facetas do *slot*. Caso contrário é mantido o valor anterior a alteração. Quando a adição de valor é confirmada o *demon if-added* é avaliado e a função retorna t. Caso contrário, é avaliado o *demon if-not-added* e o valor retornado é ().

```
? (deframe f1
        (slot s1
            (card-max 2)
            (default-v 10)
            (if-added (setq r 'sim))
            (if-not-added (setq r 'nao))))
= f1
? (setq a (new 'f1))
= #<object f1 s1 #{10}>
? (add-value 11 's1 a)
= t
? a
= #<object f1 s1 #{11 10}>
? r
= sim
? (add-value 12 's1 a)
= ()
? a
= #<object f1 s1 #{11 12}>
? r
= nao
```

(add-value2 <s> <symb> <inst>) **[SUBR]**

Tem o mesmo objetivo que a função *add-value*. A diferença é que esta função não leva em consideração a existência de facetas ou dos *demons if-added/if-not-added*. Esta função é normalmente usada quando a instância tem a mesma semântica dos objetos clássicos da programação orientada a objeto.

(remove-value <s> <symb> <inst>) [SUBR]

Remove o valor <s> do conjunto de valores já existentes para o *slot* <symb> da instância <inst>. O valor só será removido se o conjunto de valores resultante desta remoção passar pelas restricções definidas pelas facetas do *slot*. Caso contrário é mantido o valor anterior a alteração. Quando a remoção de valor é confirmada o *demon if-removed* é avaliado e a função retorna t. Caso contrário, é avaliado o *demon if-not-removed* e o valor retornado é (). A remoção de um valor que não pertence ao conjunto de valores do *slot* é sempre considerada válida.

Por exemplo, seja o *frame* "Armazem" contendo um *slot* chamado "estoque" cujos valores são peças p_i que podem ser armazenadas. Os estoques mínimo e máximo são dados pelas facetas *card-min* e *card-max*, respectivamente. Os *demons if-removed* e *if-not-removed* têm os objetivos de, respectivamente, registrar o estoque restante e enviar uma mensagem alertando que a remoção não pode ser realizada por causa da restrição do estoque mínimo.

```
? (deframe Armazem
        (slot estoque
            (default-v #{p1 p2})
            (card-min 2)
            (card-max 5)
            (if-added (print "Novo estoque : " ^estoque))
            (if-not-added (print "Armazem lotado !"))
            (if-removed (print "Estoque restante : " ^estoque))
            (if-not-removed (print "Remocao invalida !"))))

= Armazem
?
? (setq arm1 (new 'Armazem))
= #<object Armazem estoque #{p1 p2}>
?
? (add-value #{p3 p4} 'estoque arm1)
Novo estoque : #{p1 p2 p3 p4}
= t
?
? arm1
= #<object Armazem estoque #{p1 p2 p3 p4}>
?
? (add-value #{p5 p6} 'estoque arm1)
Armazem lotado !
= ()
?
? arm1
= #<object Armazem estoque #{p1 p2 p3 p4}>
?
? (remove-value p3 'estoque arm1)
Estoque restante : #{p1 p2 p4}
= t
```

```
?
? arm1
= #<object Armazem estoque #{p1 p2 p4}>
?
? (remove-value #{p4 p2} 'estoque arm1)
Remocao invalida !
= ()
?
? arm1
= #<object Armazem estoque #{p1 p2 p4}>
? (add-value2 #{a b c d e} 'estoque arm1)
= t
? arm1
= #<object Armazem estoque #{p1 p2 p4 a b c d e}>
? (remove-value2 #{b c e d p1 p2 p4} 'estoque arm1)
= t
? arm1
= #<object Armazem estoque #{a}>
? (remove-value2 a 'estoque arm1)
= t
? arm1
= #<object Armazem estoque ()>
```

(remove-value2 <valor> <instância>) **[SUBR]**

Tem o mesmo objetivo que a função *remove-value*. A diferença é que esta função não leva em consideração a existência de facetas ou dos *demons if-removed/if-not-removed*. Esta função é normalmente usada quando a instância tem a mesma semântica dos objetos clássicos da programação orientada a objeto.

(call-method <sym1> <inst> <s1> ... <sN>) **[SUBR]**

Aplica o método de nome <sym1> sobre a instância <inst> com os argumentos <si>. Quando um método é aplicável sobre a própria instância que o define a identificação desta instância é feita através da expressão *(itself)*.

```
? (deframe f1
        (slot s1
            (default-v 2))
        (slot s2
            (default-v 4))
        (demethod soma-s1-s2 () (+ ^s1 ^s2)))
= f1
```

```
? (setq a (new 'f1))
= #<object f1 s1 2 s2 4>
? (call-method 'soma a)
= 6
```

(retry) [FSUBR]

Quando um *slot* é preenchido com um valor inválido durante a instanciação a função *retry* faz com que este valor seja descartado e o *slot* seja preenchido novamente. Este ciclo repete-se até que seja atribuído um valor válido para o *slot*. Esta função só tem efeito se estiver contida no *demon if-not-filled*.

```
? (deframe Homem
        (slot idade
            (value-r (member-of ^idade [0 150]))
            (if-needed (read-value "Idade ?"))
            (if-not-filled (print "Valor invalido !")
                           (retry))))
= Homem
?
? (make 'Homem)
Idade ? 354
Valor invalido !
Idade ? 5
= #<frame Homem idade 5>
```

(confirm) [FSUBR]

Tem o mesmo objetivo da função *retry*. Difere apenas no fato de que o usuário é solicitado a confirmar o valor inválido atribuído ao *slot*.

```
? (deframe Homem
        (slot idade
            (value-r (member-of ^idade [0 150]))
            (if-needed (read-value "Idade ?"))
            (if-not-filled (print "Valor invalido !")
                           (confirm))))
= Homem
?
? (make 'Homem)
Idade ? 354
Valor invalido !
Confirma valor do slot <s/n> ? s
= ()
```

```
?
? (make 'Homem)
Idade ? 354
Valor invalido !
Confirma valor do slot <s/n> ? n
Idade ? 24
= #<frame Homem idade 24>
```

2.3 Funções Relativas aos Tipos Criados

(integer <s>) [SUBR]

Retorna <s> se este argumento for um número inteiro. Caso contrário retorna ()

```
? (integer 'bola)
= ()
? (integer 565)
= 565
? (integer 4.5)
= ()
? (integer '(3 4))
= ()
```

(real <s>) [SUBR]

Retorna <s> se este argumento for um número real. Caso contrário retorna ().

```
? (real 'bola)
= ()
? (real 565)
= ()
? (real 4.5)
= 4.5
? (real '(3 4))
= ()
```

(Set <s>) **[SUBR]**

Transforma <s> no tipo *Set*. Este tipo é usado para representar um conjunto de valores associados a um *slot*.

```
? (Set 5)
= #{5}
? (Set 'bola)
= #{bola}
? (Set '(a b c))
= #{(a b c)}
```

(Setp <s>) **[SUBR]**

Retorna t se <s> for do tipo *Set*. Caso contrário retorna ().

```
? (Setp 5)
= ()
? (Setp '(a b))
= ()
? (Setp #{5})
= t
```

(Setdel <s> <set>) **[SUBR]**

Remove <s> do conjunto de valores <set>.

```
? (Setdel 'a #{a b c})
= #{b c}
? (Setdel #{a c} #{a b c})
= #{b}
? (Setdel #{a b} #{a b})
= ()
? (Setdel 'd #{a b c})
= #{a b c}
? (Setdel #{e f g} #{a b})
= #{a b}
? (Setdel 'a #{b a c a})
= #{b c a}
```

(Setfirst <set>) **[SUBR]**

Retorna o primeiro elemento do conjunto <set>.

```
? (Setfirst #{a b c})
= a
? (Setfirst #{(a b) c})
= (a b)
? (Setfirst #{a})
= a
```

(Setcons <s> <set>) **[SUBR]**

Inclui <s> no conjunto <set>.

```
? (Setcons 'a #{b c})
= #{a b c}
? (Setcons '(a b c) #{4})
= #{(a b c) 4}
? (Setcons 'a #{a})
= #{a a}
```

(Setlast <set>) **[SUBR]**

Retorna o último valor do conjunto <set>.

```
? (Setlast #{a b c})
= c
? (Setlast #{ultimo})
= ultimo
? (Setlast #{a b #{c d}})
= #{c d}
```

(Setcdr <set>) **[SUBR]**

Retorna o conjunto <set> sem o primeiro elemento.

```
? (Setcdr #{a b c})
= #{b c}
? (Setcdr #{a})
= ()
```

(Setmember <s> <set>) **[SUBR]**

Retorna t se <s> pertence ao conjunto <set>. Caso contrário retorna ().

```
? (Setmember 'a #{a b c})
= t
? (Setmember 'i #{a b c})
= ()
? (Setmember #{a} #{a b c})
= ()
? (Setmember #{a} #{a b {a} c})
= t
```

(Setvalue <s1> ... <sN>) **[SUBR]**

Retorna um conjunto do tipo *Set* cujos elementos são o resultado da avaliação das expressões <si>.

```
? (Setvalue '(+ 4 5) 7 '(car '(a b c)))
= #{9 7 a}
```

2.4 Funções Gerais

(clear-mem) **[FSUBR]**

Remove todos os frames da memória. Retorna sempre t.

```
? (deframe f1)
= f1
? (framep 'f1)
= f1
? (clear-mem)
= t
? (framep 'f1)
= ()
```

(minimum <num1> ... <numN) **[SUBR]**

Retorna o valor do menor número <numi>. <numi> pode ser do tipo integer ou real.

? (minimum 3 4 5 2.3 5)
= 2.3
? (minimum 1 8 2 0)
= 0

(maximum <num1> ... <numN) **[SUBR]**

Retorna o valor do maior número <numi>. <numi> pode ser do tipo integer ou real.

? (maximum 3 4 5 2.3 5)
= 5
? (maximum 1 8.3 8 2 0)
= 8.3

(member-of <s> <intervalo>) **[SUBR]**

Retorna t se <s> pertence ao <intervalo>. Caso contrário retorna (). O <intervalo> pode ser contínuo ou discreto. O símbolo "+" representa intervalo infinito à direita e o símbolo "-" representa intervalo infinito à esquerda.

? (member-of 5 [0 20])
= t
? (member-of 22 [10 22[)
= ()
? (member-of 'a {e d a k})
= t
? (member-of 4.3 [1.2 5.3])
= t
? (member-of 'oi {agua pato})
= ()

(ltax <symb>) **[FSUBR]**

Lê o arquivo de nome <symb> contendo declarações de *frames*. O sufixo do arquivo deve ser ".TXT".

? (ltax arq1)
= arq1

APÊNDICE 3

EXEMPLOS DE UTILIZAÇÃO DA LINGUAGEM

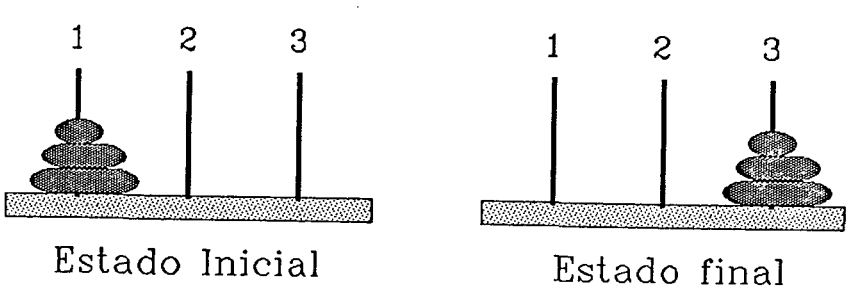
Este apêndice contém diversos exemplos de utilização da linguagem LOF. O seu objetivo é enfatizar as características e estilo de programação orientado a *frames*.

Cada exemplo contém uma breve descrição do problema e dos aspectos ligados a sua formulação e resolução.

1. Torres de Hanoi

O problema da torres de Hanoi é um problema clássico de Inteligência Artificial cuja resolução se presta bem a uma abordagem por decomposição do problema [GHALLAB 87].

Sejam três pinos verticais. Sobre o primeiro são empilhados N discos por ordem de diâmetro estritamente decrescente. O problema consiste em obter o mesmo empilhamento sobre o terceiro pino deslocando os discos um por um e sem nunca empilhar um disco em cima de um disco de diâmetro menor.



Estado Inicial Estado final

A estratégia de resolução por decomposição consiste em transformar este problema nos três subproblemas mais simples a serem resolvidos sequencialmente: 1) mover N-1 discos do pino 1 para o pino 2; 2) então mover o disco N para o pino 3; 3)e então mover os N-1 discos do pino 2 para o pino 3. Este ciclo é repetido até que todos os discos encontrem-se no pino 3.

A modelagem do problema em LOF é feita no *frame* "Hanoi". Este *frame* é composto de dois tipos de informação: de um lado informações sobre o estado do problema (*slots* "discos", "pino-orig", "pino-dest" e "pino-aux"); do outro sobre sua resolução (*slots* "sp1", "sp2" e "sp3"). A descrição dos *slots* em questão é dada abaixo.

- discos : contém o número de discos do problema;
- pino-orig : contém o nome do pino de origem;
- pino-dest : contém o nome do pino de destino;
- pino-aux : contém o nome do pino aux;
- sp1 : representa o subproblema 1;
- sp2 : representa o subproblema 2;
- sp3 : representa o subproblema 3.

A resolução é conduzida pela criação de uma instância do tipo *frame*. O preenchimento dos seus *slots* determina a sequência de ações a serem executadas para resolver o problema (na ordem sp1, sp2 e sp3).

```
? (deframe Hanoi
        (slot discos
           (value-r (> ^discos 0))
           (if-needed (prinflush "No. de discos : ")
                      (read)) )
        (slot pino-orig
           (if-needed (prinflush "pino origem : ")
                      (read)) )
        (slot pino-dest
           (if-needed (prinflush "pino destino : ")
                      (read)) )
        (slot pino-aux
           (if-needed (prinflush "pino auxiliar : ")
                      (read)) )
```

```
        (slot sp1
           (if-needed t)
           (if-filled (make 'Hanoi
                            'discos (1- ^discos)
                            'pino-orig ^pino-orig
                            'pino-dest ^pino-aux
                            'pino-aux ^pino-dest)) )
        (slot sp2
           (if-needed (incr cc)
                            (prinflush cc " : ")
                            (print (catenate "Levar o disco "
                            ^discos " de "
                            ^pino-orig " para "
                            ^pino-dest))) )
        (slot sp3
           (if-needed t)
           (if-filled (make 'Hanoi
                            'discos (1- ^discos)
                            'pino-orig ^pino-aux
                            'pino-dest ^pino-dest
                            'pino-aux ^pino-orig)))
= Hanoi
```

A função hh facilita a entrada dos dados iniciais.

```
? (de hh (n origem destino ponte)
        (make 'Hanoi
                'discos n
                'pino-orig origem
                'pino-aux ponte
                'pino-dest destino)
        (setq cc 0)
t)
= ff
```

A variável cc é usada apenas como contador de movimentos.

```
? (setq cc 0)
= 0
```

A seguir tem-se um exemplo de utilização do *frame* "Hanoi" onde propõe-se solução do problema para mover três discos do pino A para o pino B usando o pino C.

```
? (hh 3 'A 'B 'C)
1 : Levar o disco 1 de A para B
2 : Levar o disco 2 de A para C
3 : Levar o disco 1 de B para C
4 : Levar o disco 3 de A para B
5 : Levar o disco 1 de C para A
6 : Levar o disco 2 de C para B
7 : Levar o disco 1 de A para B
= t
```

2. Simulação de Circuitos Digitais

Este exemplo é dado com o objetivo de reforçar o papel dos *demons* nos processos de utilização dos *frames*. Circuitos digitais são circuitos compostos de Portas Lógicas (PL) (*and*, *or*, *not*, etc.) interligadas. Eles contêm vários pontos de entrada de corrente e um ou mais pontos de saída. Através da simulação é possível testar circuitos sem ter que construí-los fisicamente.

A simulação de circuitos lógicos usando LOF permite explorar a combinação das características de programação orientada a *frames* e a objetos. Cada PL é representada por um *frame*. No exemplo apresentado aqui, estão definidos os *frames* "Porta_Logica", "And2", "Or2" e "Not".

O *frame* "Porta_Logica" contém informações comuns a todos os tipos de PL e é representado pelos seguintes *slots*:

- nome-saida : contém o nome da saída da PL;
- saida : contém o valor lógico da saída da PL. Quando este *slot* é alterado, o novo valor é exibido na tela e, se esta saída estiver ligada à outra PL, o novo valor é passado para ela;
- lig-porta : contém o nome da PL a qual a sua saída está ligada. Por *default* a porta está isolada (representada pelo símbolo "terra");
- lig-entrada : contém o nome da entrada da PL contida no *slot* "lig-porta". Por *default* a saída está aterrada (representada pelo símbolo "terra");

Existem dois métodos associados ao *frame* "Porta_Logica":

- liga_portas que permite ligar duas PL;

- sensibiliza_entrada que introduz um valor (0 ou 1) em uma das entradas da PL.

O *frame* "And2" (e "Or2") é uma subclasse do *frame* "Porta_Logica" e representa a PL *and* (e *or*) de duas entradas e uma saída. Além dos *slots* herdados da sua superclasse

contém os *slots* "e1" e "e2" representando as suas duas entradas. Quando o valor de um destes *slots* é alterado o resultado da operação lógica *and* (e *or*) entre os valores dos dois *slots* é afetado (via ação reflexa do *demon if-filled*) ao *slot* "saida".

O *frame* "Not" é uma subclasse do *frame* "Porta-Logica" e representa a PL *not* de uma entrada e uma saída. Além dos *slots* herdados da sua superclasse contém os *slots* "saida" que contém o valor lógico de saída da PL (cuja declaração neste *frame* deve-se unicamente a redefinição do seu valor por *default*) e "e" representando a sua entrada. Quando o valor deste *slot* é alterado o resultado da operação lógica *not* sobre o novo valor é colocado (via ação reflexa do *demon if-filled*) no *slot* "saida".

A seguir a declaração em LOF dos *frames* definidos acima é apresentada. As funções "or-bin", "and-bin" e "not-bin" representam, respectivamente, as operações lógicas "or", "and" e "not" aplicadas a dois valores.

```
; Funções auxiliares
; -----------------

(de or-bin (n1 n2) (if (eq 0 (+ n1 n2)) 0 1))
(de and-bin (n1 n2) (if (eq 2 (+ n1 n2)) 1 0))
(de not-bin (n) (if (eq 1 n) 0 1))


; Declaração do frame Porta-Logica
; --------------------------------

(deframe Porta_Logica
        (slot nome-saida
            (if-needed (read-value "Qual o nome da saida")) )

        (slot saida
            (default-v 0)
            (if-filled (if (equal 'terra ^lig-porta)
                    (print "Saida " ^nome-saida " = "
                                        ^saida)
                    (put-value ^saida ^lig-entrada (eval
                                ^lig-porta)))) )

        (slot lig-porta
            (default-v 'terra) )
```

```
        (slot lig-entrada
            (default-v 'terra) )

        (demethod liga_portas (nome-inst entrada)
            (put-value nome-inst 'lig-porta (itself))
            (put-value entrada 'lig-entrada (itself)))

        (demethod sensibiliza_entradas (valor entrada)
            (put-value valor entrada (itself)))
)



; Declaração do frame And2
; ------------------------

(deframe And2
        (subclass-of Porta_Logica)
        (slot e1
            (default-v 0)
            (if-filled (put-value (and-bin ^e1 ^e2) 'saida
                    (itself))))

        (slot e2
            (default-v 0)
            (if-filled (put-value (and-bin ^e1 ^e2) 'saida
                    (itself))))
)


; Declaração do frame Or2
; -----------------------

(deframe Or2
        (subclass-of Porta_Logica)
        (slot e1
            (default-v 0)
            (if-filled (put-value (or-bin ^e1 ^e2) 'saida
                    (itself))))

        (slot e2
            (default-v 0)
            (if-filled (put-value (or-bin ^e1 ^e2) 'saida
                    (itself))))
)
```

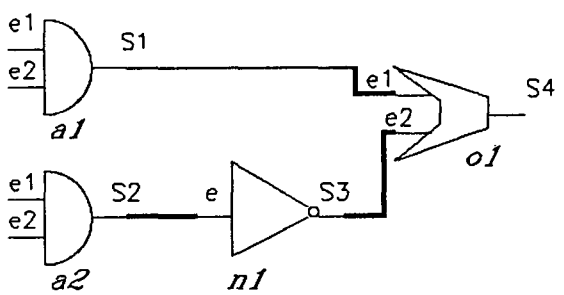
```
; Declaração do frame Not
; ----------------------

(deframe Not
        (subclass-of Porta_Logica)
        (slot saida
            (default-v 1) )

        (slot e
            (default-v 0)
            (if-filled (put-value (not-bin ^e) 'saida
                    (itself))))

)
```

A representação do circuito mostrado abaixo é feita da seguinte forma:




```
; Portas do Circuito :

? (setq a1 (new 'And2 'nome-saida 'S1))
= #<object And2 e1 0 e2 0 nome-saida S1 saida 0 lig-porta terra lig-entrada terra>
? (setq a2 (new 'And2 'nome-saida 'S2))
= #<object And2 e1 0 e2 0 nome-saida S2 saida 0 lig-porta terra lig-entrada terra>
? (setq n1 (new 'Not  'nome-saida 'S3))
= #<object Not saida 1 e 0 nome-saida S3 lig-porta terralig-entrada terra>
? (setq o1 (new 'Or2  'nome-saida 'S4))
= #<object Or2 e1 0 e2 0 nome-saida S4 saida 0 lig-porta terra lig-entrada terra>
```

```
; Ligacoes das Portas :

? (call-method 'liga_portas a1 'o1 'e1)
= t
? (call-method 'liga_portas a2 'n1 'e)
= t
? (call-method 'liga_portas n1 'o1 'e2)
= t
```

Uma sessão de utlização destes *frames* é dada no diálogo abaixo:

```
? (call-method 'sensibiliza_entrada a1 1 'e1)
Saida S4 = 0
= t
? a1
= #<object And2 e1 1 e2 0 nome-saida S1 saida 0 lig-porta o1 lig-entrada e1>
? a2
= #<object And2 e1 0 e2 0 nome-saida S2 saida 0 lig-porta n1 lig-entrada e>
? n1
= #<object Not saida 1 e 0 nome-saida S3 lig-porta o1 lig-entrada e2>
? o1
= #<object Or2 e1 0 e2 0 nome-saida S4 saida 0 lig-porta terra lig-entrada terra>
? (call-method 'sensibiliza_entrada a1 1 'e2)
Saida S4 = 1
= t
? a1
= #<object And2 e1 1 e2 1 nome-saida S1 saida 1 lig-porta o1 lig-entrada e1>
? a2
= #<object And2 e1 0 e2 0 nome-saida S2 saida 0 lig-porta n1 lig-entrada e>
? n1
= #<object Not saida 1 e 0 nome-saida S3 lig-porta o1 lig-entrada e2>
? o1
= #<object Or2 e1 1 e2 0 nome-saida S4 saida 1 lig-porta terra lig-entrada terra>
? (call-method 'sensibiliza_entrada a2 1 'e2)
Saida S4 = 1
= t
? a1
= #<object And2 e1 1 e2 1 nome-saida S1 saida 1 lig-porta o1 lig-entrada e1>
? a2
= #<object And2 e1 0 e2 1 nome-saida S2 saida 0 lig-porta n1 lig-entrada e>
? n1
= #<object Not saida 1 e 0 nome-saida S3 lig-porta o1 lig-entrada e2>
```

```
? o1
= #<object Or2 e1 1 e2 1 nome-saida S4 saida 1 lig-porta terra lig-entrada terra>
```

3. Representação Consistente de uma Data

A representação consistente de uma data através de um *frame* LOF permite enfatizar a importância das facetas e dos *demons* com relação a representação do conhecimento. Para este exemplo, uma data é formada pelo ano, mês, dia do mês, dia da semana, hora e minuto. O *frame* "DATA" é composto dos seguintes *slots*:

- Ano, que deve conter um número inteiro maior que zero;
- Mes, que deve conter um número inteiro entre 1 e 12 e que seja consistente com o valor do dia do mês;
- DiaMes, que deve conter um número inteiro entre 1 e 31 e que seja consistente com o valor do mês;
- DiaSemana, que deve conter um valor pertencente ao conjunto {seg ter qua qui sex sab dom} e estar consistente com os slots "Ano", "Mes" e "DiaMes";
- Hora, que deve conter um número inteiro entre 0 e 23;
- Minuto, que deve conter um número inteiro entre 0 e 59.

Neste exemplo, utilizou-se a faceta *unit* para orientar o usuário sobre quais os valores esperados para o *slot* a que a faceta está associada. A seguir tem-se a declaração em LOF do frame "DATA" como definido acima.

```
; Declaração do frame DATA
; ------------------------

(deframe DATA
    (slot Ano
                (unit "integer")
                (value-r (integer ^Ano)
                                (> ^Ano 0))
                (if-needed (read-value "Ano :"))
                (if-not-filled (print "Ano ilegal !!")
                                (retry)) )
```

```
(slot Mes
        (unit "1 a 12")
        (value-r (integer ^Mes)
                (member-of ^Mes [1 12])
                (or (and (equal ^DiaMes 31)
                        (member-of ^Mes {1 3 5 7 8 10
                                                12}))
                        (and (member-of ^DiaMes {29 30})
                                (member-of ^Mes [1 12])
                                (nequall ^Mes 2))
                        (and (member-of ^DiaMes [1 28])
                                (member-of ^Mes [1 12]))))
        (if-needed (read-value "Mes :"))
        (if-not-filled (print "Mes ilegal !!")
                        (retry)) )

(slot DiaMes
        (unit "1 a 31")
        (value-r (integer ^DiaMes)
                (member-of ^DiaMes [1 31])
                (or (and (equal ^Mes 2)
                                (member-of ^DiaMes [1 28]))
                        (and (member-of ^Mes {4 6 9 11})
                                (member-of ^DiaMes [1 30]))
                        (and (member-of ^Mes {1 3 5 7 8 10
                                                12})
                        (member-of ^DiaMes [1 31]))))
        (if-needed (read-value "Dia do mes :"))
        (if-not-filled (print "Dia do mes ilegal !!")
                        (retry)) )

(slot DiaSemana
        (unit "dom seg ter qua qui sex sab")
        (value-r (symbolp ^DiaSemana)
                (member-of ^DiaSemana
                                {seg ter qua qui sex sab
                                        dom}))
        (if-needed (read-value "Dia da semana :"))
        (if-not-filled (print "Dia da semana ilegal !!")
                        (retry)) )

(slot Hora
        (unit " 0 a 23")
        (value-r (integer ^Hora)
                (> = ^Hora 0)
                (< = ^Hora 23))
        (if-needed (read-value "Entre com a hora :"))
        (if-not-filled (print "Hora ilegal !!")
                        (retry)) )
```

```
    (slot Minuto
            (unit "0 a 59")
            (value-r (integer ^Minuto)
                        (> = ^Minuto 0)
                        (< = ^Minuto 59))
            (if-needed (read-value "Entre com o minuto :"))
            (if-not-filled (print "Minuto ilegal !!")
                                    (retry)) )
)
```

Um exemplo de criação de uma instância deste *frame* é apresentado a seguir. Nele os valores fornecidos pelo usuário estão em negrito.

```
? (make 'DATA)
Ano : [integer] 1990
Mes : [1 a 12] 2
Dia do mes : [1 31] 27
Dia da semana : [dom seg ter qua qui sex sab] ter
Hora : [0 23] 16
Minuto : [0 59] 50
= #<frame DATA Ano 1990 Mes 2 DiaMes 27 DiaSemana ter Hora 16 Minuto 50>

? (make 'DATA)
Ano : [integer] 1990
Mes : [1 12] 2
Dia do mes : [1 31] 31
Dia do mes ilegal !!
Dia do mes : [1 31] 23
Dia da semana : [dom seg ter qua qui sex sab] ter
Hora : [1 23] 34
Hora ilegal !!
Hora : [1 23] 12
Minuto : [1 59] 50
= #<frame DATA Ano 1990 Mes 2 DiaMes 23 DiaSemana ter Hora 12 Minuto 50>
```

4. Criação de Um Objeto do Tipo Menu

Um menu é uma interface homem-máquina agradável para fornecer informações (através da escolha de uma ou mais opções). A implementação do *frame* "Menu" apresentada a seguir permite enfatizar as características de programação orientada a objetos em LOF. Uma instância deste *frame* é criada para armazenar o título e a lista de opções do menu. Sua utilização é feita através da invocação de métodos que o visualisam na tela de modo que o usuário possa escolher uma das opções. Para isso o *frame* é composto do seu título e das opções oferecidas (respectivamente *slots* "titulo" e "opcoes").

O *frame* "Menu" é formado também pelos seguintes métodos:

- desenha : desenha a moldura do menu;
- apaga : apaga o menu da tela após ter sido escolhida uma das opções;
- controla : controla o processo de escolha de uma opção que é feito da seguinte forma: pressionando as teclas correspondentes as setas para cima e para baixo é mudada a opção corrente. Pressionando a tecla enter é selecionada a opção corrente;
- escolhe : método invocado para utilizar a instância *frame* "Menu". Este método invoca os métodos citados acima e retorna a opção escolhida. Ele possui como parâmetros, além da instância, dois valores numéricos correspondentes às coordenadas da tela.

As funções e o *frame* que implementam o objeto "menu" descrito acima são dados a seguir.

```
; Funções auxiliares
; -----------------

(de desenha-quadro (cab ci li cf lf compr)
    (let ((a (explode (makestring (- cf (1- ci)) 196)))
          (br (explode (makestring (1+ compr) 32)))
         )
         (apply 'tyco (cons ci (cons li a)))
         (apply 'tyco (cons (1+ ci) (cons li (explode cab))))
         (for (i (1+ li) 1 (1- lf))
              (apply 'tyco (cons ci (cons i br)))
              (tyco ci i 179)
              (tyco cf i 179)
         )
         (apply 'tyco (cons ci (cons lf a)))
         (tyco ci li 218)
         (tyco ci lf 192)
         (tyco cf li 191)
         (tyco cf lf 217)
         )
)


(de maior-compr (l)
    (let ((m 0))
        (while l
              (if (> (slength (car l)) m)
                  (setq m (slength (car l)))
              )
              (nextl l)
        )
        m
))


; Declaração do frame Menu
; ------------------------

(deframe Menu

    (slot titulo
        (if-needed (read-value "Nome do menu : ")) )

    (slot opcoes
        (if-needed (read-value "Lista de Opcoes : ")) )
```

```
(demethod desenha (ci li)
        (let ((op ^opcoes))
            (desenha-quadro ^titulo
                            ci
                            li
                            (+ ci (maior-compr op) 1)
                            (+ li (length op) 1)
                            (maior-compr op))
            (for (i (1+ li) 1 (+ li (length op)))
                (tycursor (1+ ci) i)
                (print (nextl op)))))

(demethod apaga (ci li)
        (let ((tam-col (+ 2 (maior-compr ^opcoes)))
                (tam-lin (1+ (length ^opcoes)))
                (str)
                )
                (setq str (explode (makestring tam-col
                                                        32)))
                (for (i 0 1 tam-lin)
                    (apply 'tyco (cons ci (cons (+ i li)
                                                        str)))
                )
        ))

(demethod escolhe (ci li)
        (let ((r))
                (call-method 'desenha (itself) ci li)
                (setq r (call-method 'controla (itself)
                                                        ci li))
                (call-method 'apaga (itself) ci li)
                (tycursor 0 li)
                r
        ))

(demethod controla (ci li)
        (let ((r 0)
                (tecla)
                (pi (1+ ci))
                (li (1+ li))
                (tam-lin (length ^opcoes))
                )
                (tycursor pi li)
                (while (nequal (setq tecla (tyi)) 13)
                        (when (and (equal 0 tecla) (setq tecla
                                                        (tyi)))
                          (selectq tecla
                            (80         ;desce
                              (when (< (1+ r) tam-lin)
                                (incr r)
                                (tycursor pi (+ li r))
                              ))
```

```
                                (72             ;sobe
                                 (when (> r 0)
                                    (decr r)
                                    (tycursor pi (+ li r))
                                 ))
                            )
                        )
                    )
                    (nth r ^opcoes)
                    ))
)
```

Uma sessão de utilização deste *frame* é apresentada a seguir. A opção escolhida pelo usuário é marcada com "*".

```
? (setq m (make 'Menu))
Titulo : Frutas
Opcoes : (maca pera abacate)
= #<frame Menu titulo Frutas opcoes (maca pera abacate)>

? (call-method 'escolhe m 0 3)
-Frutas--
|maca   |
|pera   |*
|abacate|
---------

= pera
```

5. Uma Taxonomia de Computadores

O exemplo a seguir visa ilustrar o processo de classificação em LOF. Define-se, para isso, uma taxonomia simples entre modelos de computadores cujo objetivo é puramente acadêmico.

A construção de uma taxonomia de computadores permite explorar o mecanismo de classificação em LOF. O problema consiste em determinar a que classe de computadores definidos nessa taxonomia pertence um computador particular definido no contexto da classificação.

Neste exemplo, a taxonomia é composta por cinco *frames* representando computadores hipotéticos dispostos em três níveis. O primeiro nível contém o *frame* "Computador" representando informações válidas para qualquer tipo de computador. O segundo nível contém os *frames* "PC" e "Workstation" que representam duas linhas de computadores. O terceiro nível contém os *frames* "Cobra_XT" e "Sun-4/260" que representam dois tipos de computadores ligados, respectivamente, aos *frames* "PC" e "Workstation".

O *frame* "Computador" é formado pelos seguintes *slots*:

- mem : contém o valor de memória principal. Este valor (em Kbytes) deve pertencer ao conjunto {512 640 704 1024 2048 4096 8192};
- vel-proc : contém o valor da velocidade de processamento em MIPS, que deve estar entre 0.01 e 100;
- mem-sec : contém o valor de memória secundária. Este valor (em Mbytes) deve pertencer ao conjunto {20 30 50 100 200};

O *frame* "PC" é formado pelos seguintes *slots*:

- vel-proc : o valor deste *slot* deve estar entre 0.01 e 1.5;
- marca : o valor deste *slot* permite a escolha de uma marca de fabricante. As escolhas possíveis são "Cobra" ou "SID";

O *frame* "Cobra_XT" é formado pelos seguintes *slots*:

- marca : o valor deste *slot* deve ser "Cobra";
- mem : o valor deste *slot* deve pertencer ao conjunto {512 640 704 1024};
- mem-sec : o valor deste *slot* deve pertencer ao conjunto {20 30};
- processador : contém o processador usado. O único valor adimitido é 8088;
- preco : contém o preço do computador em função da configuração desejada. O valor deste *slot* é dado pelo seguinte cálculo: mem * 4.12 + mem-sec * 3 + vel-proc * 30;

O *frame* "Workstation" é formado pelos seguintes *slots*:

- vel-proc : o valor deve estar entre 1 e 10;
- processador : contém o processador usado. O valor deve pertencer ao conjunto {80386 68000 SPARC};

O *frame* "Sun-4/260" é formado pelos seguintes *slots*:

- processador : o único valor possível é 68000;

- preco : contém o preço do computador em função da configuração desejada. O valor deste slot é dado pelo seguinte cálculo: mem * 9.5 + mem-sec * 7.2 + vel-proc * 55.2;

A seguir tem-se a declaração dos frames acima descritos.

```
; Declaração do frame Computador
; ------------------------------

(deframe Computador
    (slot mem
        (unit "Kb")
        (if-needed (call-method 'escolhe
                                    (make 'Menu
                                      'titulo 'Mem
                                      'opcoes '(512 640
                                                704 1024
                                                2048 4096
                                                8192))
                                      40 0)) )

    (slot vel-proc
        (unit "MIPS")
        (value-r (member-of ^vel-proc [0.01 100]))
        (if-needed (read-value "Qual a velocidade de
                                    processamento ?")) )

    (slot mem-sec
        (unit "Mb")
        (if-needed (call-method 'escolhe
                                    (make 'Menu
                                            'titulo 'Sec
                                            'opcoes '(20 30 50
                                                        100 200))
                                      40 0)) )
)

; Declaração do frame PC
; ----------------------------

(deframe PC
    (subclass-of Computador)

    (slot vel-proc
        (value-r (member-of ^vel-proc [0.01 1.5])) )

    (slot marca
        (value-r (member-of ^marca {Cobra SID}))
        (if-needed (call-method 'escolhe
                                    (make 'Menu
                                            'titulo 'Marca
                                            'opcoes '(Cobra
                                                        SID))
                                40 0))
        (if-not-filled (print "Marca nao conhecida !")) )
)
```

```
; Declaração do frame Cobra_XT
; -------------------------

(deframe Cobra_XT
    (subclass-of PC)

    (slot marca
        (value 'Cobra) )

    (slot mem
        (value-r (member-of ^mem {512 640 704 1024})) )

    (slot mem-sec
        (value-r (member-of ^mem-sec {20 30})) )

    (slot processador
        (value 8088) )

    (slot preco
        (if-needed (+ (* ^mem 4.12)
                      (* ^mem-sec 3)
                      (* ^vel-proc 30))) )

    (if-instantiated  (print "Instanciado Cobra_XT"))
)


; Declaração do frame Workstation
; -------------------------------

(deframe Workstation
    (subclass-of Computador)

    (slot vel-proc
        (value-r (member-of ^vel-proc [1.5 10])) )

    (slot processador
        (value-r (member ^processador {80386 68000 SPARC}))
        (if-needed (call-method 'escolhe
                                (make 'Menu
                                      'titulo 'Proc
                                      'opcoes '(80386 68000
                                                SPARC))
                                40 0)) )
)
```

```
; Declaração do frame Sun-4/260
; ----------------------

(deframe Sun-4/260
    (subclass-of Workstation)

    (slot processador
        (value 68000) )

    (slot preco
        (if-needed (+ (* ^vel-proc 55.20)
                      (* ^mem 9.5)
                      (* ^mem-sec 7.2))) )

    (if-instantiated (print "Instanciado Sun-4/260"))
)
```

Uma sessão de utilização desta taxonomia é dada a seguir. O símbolo "*" indica a opção escolhida quando a informação é obtida via menu. As demais informações fornecidas pelo usuário estão sublinhadas.

```
? (classify 'Computador)

-Mem_Principal [Kb] -
| 512          |
| 640          |
| 704          |
| 1024         |*
| 2048         |
| 4096         |
--------------------

Qual a velocidade de processamento ? [MIPS] 10

-Mem_Secundaria [Mb] -
| 20           |
| 30           |
| 50           |*
| 100          |
| 200          |
---------------------
```

```
-Processador -
| 80386     |
| 68000     |*
| SPARC     |
-------------

Instanciado Sun-4/260 !
= t

? (all-slots 'Sun-4/260)
= (processador preco vel-proc mem mem-sec)


; valor dos slots da instância Sun-4/260

? (mapcar '(lambda (x) (slot-value x Sun-4/260))
        (all-slots 'Sun-4/260))
= (68000 10640 10 1024 50)
```

APÊNDICE 4

SISTEMA ESPECIALISTA DE DIAGNÓSTICO DE DEFEITOS EM EQUIPAMENTOS DE SUBESTAÇÃO DE ALTA TENSÃO

Este apêndice contém o conjunto de *frames* que formam um Sistema Especialista de diagnóstico de defeitos em equipamentos de Substações de Alta Tensão apresentado no capítulo 5.

```
(when (not (framep 'Menu))
      (print "Lendo arquivo MENU.TXT ...")
      (load "menu.txt")
)


;*************************************************************
;              INICIALIZACAO DA BASE DE FATOS
;*************************************************************

(deframe Inicializa
    (slot equipamentos
        (if-needed (setq 01DJ (make 'Disjuntor_Mitsu_DJX1
                                        'Código '01DJ))
                   (setq Configurador (new 'Configurador))
                   (call-method 'configura Configurador 01DJ)
                   (setq Bay_1 (make 'Bay_1))
                   (setq Subestacao (make 'Subestacao)) ))
)


;*************************************************************
;              CONFIGURADOR DE EQUIPAMENTOS
;*************************************************************


(deframe Configurador

    (demethod configura (obj)
        (let ((menu (new 'Menu)))
             (tycls)
             (tycursor 0 0)
             (print (catenate "Configuracao de "
                              (slot-value 'nome obj)))
```

```
                (print (makestring 78 205))
                (mapc '(lambda (x)
                                (put-value2
                                    (call-method 'escolhe
                                                    (and (put-value2
                                                                (car x)
                                                                'titulo
                                                                menu)
                                                            (put-value2
                                                                (cdr x)
                                                                'opcoes
                                                                menu)
                                                            menu)
                                                    0 3)
                                                    (car x)
                                    obj))
                                    (slot-value 'config obj))
                    (tycls)))
)


;*************************************************************
;                       SUBSTACAO
;*************************************************************

(deframe Subestacao
    (slot bays (value '(1 2 3 4)))
    (slot nome (value "Palhoca"))
)


;*************************************************************
;       DESCRICAO DA TOPOLOGIA DE UMA SUBESTACAO
;*************************************************************


;------------
(deframe Bay_1
    (slot bay
        (value 1))
    (slot tensao
        (value 138))
    (slot nome
        (value "LT_Entrada"))
    (slot disjuntor
        (value '01DJ))
    (slot secionadora-1
        (value '01CDA))
    (slot secionadora-2
        (value '01CDB))
```

```
    (slot secionadora-3
        (value '01CDC))
    (slot equipamentos
        (value '(disjuntor
                    secionadora-1
                    secionadora-2
                    secionadora-3)))
)



;*****************************************************************
;
;              CONTROLE PRINCIPAL
;*****************************************************************

;-------------------------
(deframe Analisa_Equipamento
    (to-instantiate
        (tycls)
        (tycursor 0 0)
        (print "Sistema de Analise de Subestacao
                Baseado em Frames"
                " - Versao 1.0  nov/89")
        (print (makestring 78 205))
        (print " ") )
    (slot bay
        (if-needed (call-method 'escolhe
                                    (make 'Menu
                                        'titulo
                                        "Bay do Equipamento"
                                        'opcoes
                                        (slot-value 'bays
                                            Subestacao))
                                        0 3)))
    (slot ident-bay
        (if-needed (framep (concat 'Bay_ ^bay)))

        (if-not-filled
                    (print "O bay " ^bay " ainda nao foi
                                implementado.")))
    (slot equipamento
        (if-needed (call-method 'escolhe
                                (make 'Menu
                                    'titulo
                                    "Equipamento"
                                    'opcoes
                                    (slot-value 'equipamentos
                                        (eval ^identbay)))
                                    0 3)))
```

```
    (if-instantiated
        (selectq ^equipamento
            (disjuntor
                (let ((bay (eval ^ident-bay)))
                    (classify 'Analisa_Disjuntor
                                'Bay ^bay
                                'Tensao (slot-value 'tensao bay)
                                'Nome (slot-value 'nome bay)
                                'Disjuntor (slot-value 'disjuntor
                                                        bay))))
            (t  (print "Analise inexistente para equipamento"
                        ^equipamento))))
)



;*************************************************************
;  DESCRICAO DOS OBJETOS QUE COMPOEM UMA SUBESTACAO
;*************************************************************


;----------------
(deframe Disjuntor
    (slot Codigo
        (value-r (symbolp ^Codigo))
        (unit "symbol")
        (if-needed (read-value "Codigo do disjuntor :"))
        (if-not-filled (print "Codigo invalido. Espero um
                                    simbolo !")
                        (retry)))

    (slot Fabricante
        (value-r (member-of ^Fabricante
                                {Mitsubishi DASA Sprecher BBC
                                Siemens Sace}))
        (if-needed (read-value "Fabricante do
                                    disjuntor :"))
        (if-not-filled (print "Nome de fabricante
                                    invalido !")
                        (retry)))

    (slot Modelo
        (value-r (member-of ^Modelo {DJX1 DJX23 DJZX1}))
        (unit "DJX1 DJX23 DJZX1")
        (if-needed (read-value "Modelo do disjuntor :"))
        (if-not-filled (print "Modelo invalido !")
                        (retry)))
```

```
    (slot Operacionalidade
        (value-r (member-of ^Operacionalidade
                                {"Normal" "Nao Opera"
                                "Nao Abre" "Nao Fecha"}))
        (default-v "Normal"))
)


;----------------------
(deframe Disjuntor_Mitsu
    (subclass-of Disjuntor)

    (slot Fabricante (value 'Mitsubishi))
)

;--------------------------
(deframe Disjuntor_Mitsu_DJX1
    (subclass-of Disjuntor_Mitsu)

    (slot config
        (value '((Contato_49M 0 1)
        (Contato_8SA 0 1)
        (Contato_8A 0 1)
        (Contato_8DC2 0 1)
        (Contato_8DC1 0 1)
        (Contato_63GL 0 1)
        (Contato_63AL 0 1)
        (Contato_63AA 0 1)
        (Contato_63GA 0 1)
        (Operacionalidade "Nao Opera"
                                "Nao Abre"
                                "Nao Fecha"
                                "Normal"))))

    (slot nome
        (value "Disjuntor Mistubishi DJX1"))

    (slot Modelo (value 'DJX1))
    (slot Contato_63GA (default-v 0))
    (slot Contato_63AA (default-v 0))
    (slot Contato_63AL (default-v 0))
    (slot Contato_63GL (default-v 0))
    (slot Contato_8DC1 (default-v 0))
    (slot Contato_8DC2 (default-v 0))
    (slot Contato_8A (default-v 0))
    (slot Contato_8SA (default-v 0))
    (slot Contato_49M (default-v 0))
)
```

```
;--------------------
(deframe Seccionadora

    (slot Codigo
        (value-r (symbolp ^Codigo))
        (if-needed (prinflush "Codigo da seccionadora : ")
                                    (read))
        (if-not-filled (print "Codigo invalido. Espero um
                                        simbolo !")
                                    (retry)))
    (slot Fabricante
        (value-r (member-of ^Fabricante {Lorenzetti Hubbel}))
        (if-needed (prinflush "Fabricante da
                                        seccionadora : ")
                                (read))
        (if-not-filled (print "Nome de fabricante
                                        invalido !")
                                (retry)))

    (slot Modelo
        (value-r (member-of ^Modelo {CDMN CDMT 03CDB CDMN23
                                                CDMT23 CDMN69 CDMT69}))
        (if-needed (prinflush "Modelo da seccionadora : ")
                                (read))
        (if-not-filled (print "Modelo invalido !")
                                (retry)))
)


;--------------------
(deframe Transformador

    (slot Codigo
        (value-r (symbolp ^Codigo))
        (if-needed (prinflush "Codigo do transformador : ")
                                (read))
        (if-not-filled (print "Codigo invalido. Espero um
                                        simbolo !")
                                (retry)))

    (slot Fabricante
        (value-r (member-of ^Fabricante {Hitachi}))
        (if-needed (prinflush "Fabricante do
                                        transformador : ")
                                (read))
        (if-not-filled (print "Nome de fabricante
                                        invalido !")
                                (retry)))
```

```
    (slot Modelo
        (value-r (member-of ^Modelo {TT138-69}))
        (if-needed (prinflush "Modelo do transformador : ")
                   (read))
        (if-not-filled (print "Modelo invalido !")
                       (retry)))
)


;*************************************************************
;        TAXONOMIA QUE ANALISA DISJUNTOR
;*************************************************************


;-----------------------
(deframe Analisa_Disjuntor

    (to-instantiate
        (print "Analise de DISJUNTOR."))

    (slot Disjuntor
        (if-needed (read-value "Identificador do
                                disjuntor : ")))

    (slot Tensao
        (value-r (member-of ^Tensao {138 69 34 23 15 5}))
        (if-needed (prinflush "Tensao do disjuntor : ")
                                (read))
        (if-not-filled (print "Tensao invalida !")
                       (print "Espero um dos valores
                                {138 69 34 23 15 5}")
                       (confirm)))

    (slot Fabricante
        (value-r (member-of ^Fabricante {Sprecher Siemens
                                         Sac  Mitsubishi
                                         BBC Dasa}))
        (if-needed (slot-value 'Fabricante
                               (eval ^Disjuntor))))

    (slot Operacionalidade
        (value-r (member-of ^Operacionalidade
                            {"Nao Abre" "Nao Fecha"
                            "Normal" "Nao Opera"}))
        (if-needed (slot-value 'Operacionalidade
                               (eval ^Disjuntor))))
)
```

```
;----------------
(deframe Tensao15

    (subclass-of Analisa_Disjuntor)

    (slot Tensao
        (value-r (equal 15 ^Tensao)))

    (slot Fabricante
        (value-r (member-of ^Fabricante {Sprecher Siemens
                                                Sace}))
        (if-not-filled (print "Base de conhecimento
                                        insuficiente para
                                        tensao 15  e fabricante "
                                        ^Fabricante)))
)


;----------------
(deframe Tensao138

    (subclass-of Analisa_Disjuntor)

    (slot Tensao
        (value-r (equal 138 ^Tensao)))

    (slot Fabricante
        (value-r (member-of ^Fabricante {Sprecher
                                                Mitsubishi BBC}))
        (if-not-filled (print "Base de conhecimento
                                    insuficiente para
                                    tensao 138  e fabricante "
                                    ^Fabricante)))
)


;----------------
(deframe Tipo138_1
    (subclass-of Tensao138)
    (slot Fabricante
        (value 'Mitsubishi))
)
```

```
;-----------
(deframe MT_1
    (subclass-of Tipo138_1)

    (slot Contato_63GA
        (value-r (equal ^Contato_63GA 1))
        (if-needed (slot-value 'Contato_63GA
                                    (eval ^Disjuntor))))

    (slot Contato_63GL
        (value-r (equal ^Contato_63GL 1))
        (if-needed (slot-value 'Contato_63GL
                                    (eval ^Disjuntor))))

    (slot Operacionalidade
        (value "Nao Opera"))

    (slot Diag
        (value "Grave Vazamento de Gas [1.0]")
        (if-filled (print "Diagnostico : " ^Diag)))

    (slot Recom
        (value "Providenciar Manutencao URGENTE")
        (if-filled (print "Recomendacao : " ^Recom)))
    (if-instantiated (print "Diagnostico de : MT_1") (print))
)


;-----------
(deframe MT_2

    (subclass-of Tipo138_1)

    (slot Contato_63AA
        (value-r (equal ^Contato_63AA 1))
        (if-needed (slot-value 'Contato_63AA
                                    (eval ^Disjuntor))))

    (slot Contato_63AL
        (value-r (equal ^Contato_63AL 1))
        (if-needed (slot-value 'Contato_63AL
                                    (eval ^Disjuntor))))

    (slot Contato_8A
        (value-r (equal ^Contato_8A 0))
        (if-needed (slot-value 'Contato_8A
                                    (eval ^Disjuntor))))

    (slot Operacionalidade
        (value "Nao Opera"))
```

```
    (slot Diag
        (value "Grave Vazamento de Ar Comprimido [1.0]")
        (if-filled (print "Diagnostico : " ^Diag)))

    (slot Recom
        (value "Providenciar Manutencao URGENTE")
        (if-filled (print "Recomendacao : " ^Recom)))

    (if-instantiated (print "Diagnostico de : MT_2") (print))
)


;-----------
(deframe MT_3

    (subclass-of Tipo138_1)

    (slot Contato_63AA
        (value-r (equal ^Contato_63AA 1))
        (if-needed (slot-value 'Contato_63AA
                                    (eval ^Disjuntor))))

    (slot Contato_63AL
        (value-r (equal ^Contato_63AL 1))
        (if-needed (slot-value 'Contato_63AL
                                    (eval ^Disjuntor))))
    (slot Contato_8A
        (value-r (equal ^Contato_8A 1))
        (if-needed (slot-value 'Contato_8A
                                    (eval ^Disjuntor))))

    (slot Operacionalidade
        (value "Nao Opera"))

    (slot Diag
        (value "Defeito no Circuito do Compressor [1.0]")
        (if-filled (print "Diagnostico : " ^Diag)))

    (slot Recom
        (value "Providenciar Manutencao URGENTE")
        (if-filled (print "Recomendacao : " ^Recom)))

    (if-instantiated (print "Diagnostico de : MT_3") (print))
)
```

```
;-----------
(deframe MT_4

    (subclass-of Tipo138_1)

    (slot Contato_63GA
        (value-r (equal ^Contato_63GA 1))
        (if-needed (slot-value 'Contato_63GA
                                    (eval ^Disjuntor))))

    (slot Operacionalidade
        (value "Normal"))

    (slot Diag
        (value "Vazamento de Gas [0.8]")
        (if-filled (print "Diagnostico : " ^Diag)))

    (slot Recom
        (value "Completar Reservatorio de Gas")
        (if-filled (print "Recomendacao : " ^Recom)))

    (if-instantiated (print "Diagnostico de : MT_4") (print))
)


;-----------
(deframe MT_5

    (subclass-of Tipo138_1)

    (slot Contato_63AA
        (value-r (equal ^Contato_63AA 1))
        (if-needed (slot-value 'Contato_63AA
                                    (eval ^Disjuntor))))

    (slot Contato_8A
        (value-r (equal ^Contato_8A 1))
        (if-needed (slot-value 'Contato_8A
                                    (eval ^Disjuntor))))

    (slot Operacionalidade
        (value "Normal"))

    (slot Diag
        (value "Curto-Circuito no Sistema do Compressor [1.0]
        (if-filled (print "Diagnostico : " ^Diag)))

    (slot Recom
        (value "Vistoriar Fiacao no Armario do Disjuntor")
        (if-filled (print "Recomendacao : " ^Recom)))
```

```
    (if-instantiated (print "Diagnostico de : MT_5") (print))
)


;------------
(deframe MT_6

    (subclass-of Tipo138_1)

    (slot Contato_63AA
        (value-r (equal ^Contato_63AA 1))
        (if-needed (slot-value 'Contato_63AA
                                    (eval ^Disjuntor))))

    (slot Contato_8A
        (value-r (equal ^Contato_8A 0))
        (if-needed (slot-value 'Contato_8A
                                    (eval ^Disjuntor))))

    (slot Contato_49M
        (if-needed (slot-value 'Contato_49M
                                    (eval ^Disjuntor))))
)


;--------------
(deframe MT_6-1

    (subclass-of MT_6)

    (slot Contato_49M
        (value 1))

    (slot Diag
        (card-min 3)
        (value (Setvalue """Rele 88ACM com defeito [0.9]"""
                         """Rele T com defeito [0.7]"""
                         """Contato 63GA com defeito [0.2]""""))
        (if-filled (print "Diagnostico : " ^Diag)))

    (slot Recom
        (value "Providenciar Manutencao URGENTE")
        (if-filled (print "Recomendacao : " ^Recom)))

    (if-instantiated (print "Diagnostico de : MT_6-1")
                     (print))
)
```

```
;-------------
(deframe MT_6-2

    (subclass-of MT_6)

    (slot Contato_49M
        (value 0))

    (slot Diag
        (value "Defeito no compressor [0.9]")
        (if-filled (print "Diagnostico : " ^Diag)))

    (if-instantiated (print "Diagnostico de : MT_6-2")
                     (print))

)


;------------
(deframe MT_7

    (subclass-of Tipo138_1)

    (slot Contato_8SA
        (value-r (equal ^Contato_8SA 1))
        (if-needed (slot-value 'Contato_8SA
                               (eval ^Disjuntor))))

    (slot Diag
        (value "Curto-Circuito no Sistema de Ilum. e
                Aquecim. do DJ [1.0]")
        (if-filled (print "Diagnostico : " ^Diag)))

    (slot Recom
        (value "Fazer Manutencao")
        (if-filled (print "Recomendacao : " ^Recom)))

    (if-instantiated (print "Diagnostico de : MT_7") (print))
)


;------------
(deframe MT_8

    (subclass-of Tipo138_1)

    (slot Contato_8DC1
        (value-r (equal ^Contato_8DC1 1))
        (if-needed (slot-value 'Contato_8DC1
                               (eval ^Disjuntor))))
```

```
    (slot Operacionalidade
        (value "Nao Fecha"))

    (slot Diag
        (value "Curto-Circuito no Sistema Eletrico de
                    Fecham. do DJ [1.0]")
        (if-filled (print "Diagnostico : " ^Diag)))

    (slot Recom
        (value "Fazer Manutencao")
        (if-filled (print "Recomendacao : " ^Recom)))

    (if-instantiated (print "Diagnostico de : MT_8") (print))
)


;-----------
(deframe MT_9

    (subclass-of Tipo138_1)

    (slot Contato_8DC2
        (value-r (equal ^Contato_8DC2 1))
        (if-needed (slot-value 'Contato_8DC2
                                    (eval ^Disjuntor))))

    (slot Operacionalidade
        (value "Nao Abre"))

    (slot Diag
        (value "Curto-Circuito no Sistema Elet. de Abertura
                    do DJ [1.0]")
        (if-filled (print "Diagnostico : " ^Diag)))

    (slot Recom
        (value "Fazer Manutencao")
        (if-filled (print "Recomendacao : " ^Recom)))

    (if-instantiated (print "Diagnostico de : MT_9") (print))
)
```

REFERÊNCIAS BIBLIOGRÁFICAS

[AIKINS 83]

AIKINS, J. S., "Prototypical Knowledge for Expert Systems", Artificial Intelligence, 20:163-210, 1983.

[AIKINS 85]

AIKINS, J. S., "A Representation Scheme Using Both Frames and Rules", Rule-Based Expert Systems, B. G. Buchanan and E. H. ShortLife (eds.), Addisson-Wesley, 1985.

[ALTY 84]

ALTY, J. L. & COOMBS, M. J., "Expert Systems Concepts and Examples", NCC Publications, 1984.

[BARR 86]

BARR, A., FEIGENBAUM, E. (eds.), "The Handbook of Artificial Intelligence", vol. 1, Addison-Wesley, 1986.

[BOBROW 77]

BOBROW, D. G. et alii,"GUS: A Frame-Driven Dialog System", Artificial Intelligence, 8:155-173, 1977.

[BOBROW 77b]

BOBROW, D. G. & WINOGRAD, T.,"An Overview of KRL, a Knowledge Representation Language", Cognitive Science 1(1):3-46, 1977.

[BOBROW 77c]

BOBROW, D. G & WINOGRAD, T., "Experience with KRL-0 One Cycle of a Knowledge Representation Language", 5 o. IJCAI, 1977.

[BOBROW 86]

BOBROW, D. G. et alii, "Common Loops Mergin Lisp and Object-Oriented Programming", OOPSLA '86 Proceedings, september 1986.

[BOOCH 86]

BOOCH, G., "Object-Oriented Development", IEEE Trans. on Soft. Eng., 12(2):211-221, february 1986.

[BRACHMAN 83]

BRACHMAN, R. J., FIKES, R. E., LEVESQUE, H. J., "KRYPTON: A Functional Approach to Knowledge Representation", IEEE COMPUTER:66-73, september 1983.

[BRACHMAN 85]

BRACHMAN, R. J. & SCHMOLZE, J. G., "An Overview of the KL-ONE Knowledge Representation System", Cognitive Science, 9:171-216, 1985.

[BRACHMAN 85b]

BRACHMAN, R. J. & LEVESQUE, H. J. (eds.), "Readings in Knowledge Representation", Morgan Kaufmann, 1985.

[BYLANDER 86]

BYLANDER, T. & MITTAL, S., "CRSL: A Language for Classificatory Problem Solving and Uncertainty Handling", The AI Magazine:66-77, august 1986.

[CARNOTA 88]

CARNOTA, R. J., TESZKIEWICZ, A. D., "Sistemas Expertos y Representacion del Conocimiento", EBAI, 1988.

[CASANOVA 87]

CASANOVA, M. A., GIORNO, F. A. & FURTADO, A. L., "Programação em Lógica e a Linguagem PROLOG", Edgard Blucher, 1987.

[CHAILLOUX 85]

CHAILLOUX, J., "Le Lisp de l'INRIA Le Manuel de référence", version 15, INRIA, 1985.

[CHANDRASEKARAN 83]

CHANDRASEKARAN, B., "Towards a Taxonomy Of Problem Solving Types", The AI Magazine, :9-17, winter/spring 1983.

[CHANDRASEKARAN 86]

CHANDRASEKARAN, B., "Generic Tasks is Knowledge-Based Reasoning: High-Level Building Blocks for Expert System Design", IEEE Expert:23-30, fall 1986.

[CHERUBINI 89]

CHERUBINI, M. A., et alii, "An Integrated Expert-System Builder", IEEE Software, :44-52, november 1989.

[COX 86]

COX, B. J., "Object Oriented Programming", Addison-Wesley, 1986.

[deKLEER 86]

deKLEER, J., "An Assumption-based TMS", Artificial Intelligence, 28:127-162, 1986.

[DONOVAN 84]

DONOVAN, J., "Systems programming", McGraw-Hill, 1984.

[FAHLMAN 82]

FAHLMAN, S. E., "NETL A System for Representing and Using Real-World Knowledge", 2 ed., The MIT Press, 1982.

[FARRENY 87]

FARRENY, H. & GHALLAB, M., "Éléments d'Intelligence Artificielle", Hermes, 1987.

[FIKES 81]

FIKES, R. E., "Odyssey: A Knowledge-Based Assistant", Aritificial Intelligence 16:331-361, 1981.

[FIKES 85]

FIKES, R. E. & KEHLER, T., "The Role of Frame-Based Representation in Reasoning", Communications of the ACM, 28(9):904-920, september 1985.

[GOMEZ 81]

GOMEZ, F. & CHANDRASEKARAN, B., "Knowledge Organization and Distribuition for Medical Diagnosis", IEEE Trans. on Systems, Man, and Cybernetics, 11(1):34-42, january 1981.

[HAYES-ROTH 83]

HAYES-ROTH, F., WATERMAN, D., LENAT, D. (eds.), "Building Expert Systems", Addison-Wesley, 1983.

[INTELLICORP 86]

IntelliCorp, Inc., "New Generation Knowledge System Development Tools", Phase 1 Interim Report, 1986.

[INTELLICORP 87]

IntelliCorp, Inc., "OPUS: A New Generation Knowledge Engineering Environment", Phase 1 Final Report, 1987.

[JOHNSON 86]

JOHNSON, L. & KERAVNOV, E. T. (Eds.), "PIP", Ed. Abacus Press, 1986.

[LEVINE 88]

LEVINE, R. I., DRANG, D. E. & EDELSON, B., "Inteligência Artificial e Sistemas Especialistas", McGraw-Hill, 1988.

[KAESTNER 89]

KAESTNER, C. A. A., "Contribuição ao Estudo e Desenvolvimento de Um Sistema de Regras de Produção", dissert. mestrado, DEEL/UFSC, 1989.

[KOMOSINSKI 88]

KOMOSINSKI, L. J., & GARNOUSSET, H. E., "LOF Uma Linguagem Baseada em *Frames* Para Sistemas de Conhecimento", 5 o. Simp. Brasileiro de Inteligência Artificial, outubro 1988.

[KAYSER 84]

KAYSER, D., "Examen de Diverses Methodes Utilisées en Representation des Conaissances", Congrés R.F. et A. I., jannier 1984.

[LACERDA 87]

LACERDA, P. N., "Protótipo de um Sistema Especialista para Auxílio à Operação de Subestações de Alta Tensão", dissert. mestrado, DEEL/UFSC, 1987.

[LAURENT 87]

LAURENT, J-P., THOME, F., AYEL, J. & ZIEBELIN, D., "Evaluation comparative de trois outils de développement de systèmes experts (KEE, Knowledge Craft et ART)", Revue d'intelligence artificielle, :25-53, 1987.

[LAURIERE 82]

LAURIERE, J., "Représentation et utilisation des connaissances", Techinique et Science Informatiques, 1(2):109-133, 1982.

[MAYS 87]

MAYS, E., APTÉ, C., GRIESMER, J. & KASTNER, J., "Organizing Knowledge in a Complex Financial Domain", IEEE Expert, :61-70, fall 1987.

[MINSKY 75]

MISNKY, M., "A framework for representing knowledge", The Psychology of computer vision, P.Winston (ed.), McGraw-Hill, 1975.

[NILSSON 80]

NILSSON, N. J., "Principles of Artificial Intelligence", Tioga, 1980.

[RAMAMOORTHY 88]

RAMAMOORTHY, C. V., SHEU, P. C.,"Object-Oriented Systems", IEEE Expert, :9-15, fall 1988.

[RICH 83]

RICH, E., "Artificial Intelligence", McGraw-Hill, 1983.

[ROCHE 89]

ROCHE, C. & LAURENT, J-P., "Les approches objets et le langage LRO2 (KEOPS)", Technique et Science Informatiques, 8(1):21-39, 1989.

[ROSENBENRG ??]

ROSENBERG, S., "HPRL: A Language for Building Expert Systems", s.n.t.

[SCHMOLZE 83]

SCHMOLZE, J. G. & LIPKIS, T. A., "Classification in the KL-ONE Knowledge Representation *System*", 8 o. IJCAI, 1983.

[SMITH 85]

SMITH, D. E. & CLAYTON, J. E., "Another Look at Frames", Rule-Based Expert Systems, B. G. Buchanan and E. H. ShortLife (eds.), Addisson-Wesley, 1985.

[STEFIK 79]

STEFIK, M., "An Examination of a Frame-Structured Representation System", 6 o. IJCAI, 1979.

[STEFIK 85]

STEFIK, M. & BOBROW, D. G., "Object-Oriented Programming: Themes and Variations", The AI Magazine, 6(4):40-62, 1985.

[STEFIK 86]

STEFIK, M., BOBROW, D. G., KAHN, K. M., "Integrating Acess-Oriented Programming into a Multiparadigm Environment", IEEE Software, :10-18, january 1986.

[THOMAS 89]

THOMAS, D., "What's in an Object ?", Byte, :231-240, march 1989.

[WAH 89]

WAH, B. W., et alii,"Computers for Symbolic Processing", Proc. of the IEE, 77(4):509-539, april 1989.

[WATERMAN 86]

WATERMAN, D. A., "A Guide to Expert Systems", Addison-Wesley, 1986.

[WINSTON 84]

WINSTON, P. H., "Artificial Intelligence", Addison-Wesley, 1984.

[WINSTON 89]

WINSTON, P. H., "LISP", 3.ed., Addison-Wesley, 1989.

[WOODS 86]

WOODS, W., "Important Issues in Knowledge Representation", Proc. of the IEEE, 74(10):1322-1334, october 1986.